Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Mikko Vesikkala

Visual Regression Testing for Web Applications

Master's Thesis
Espoo, April 10, 2014

Supervisor: Professor Jukka K. Nurminen
Instructor: Juhani Snellman M.Sc.

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Mikko Vesikkala

Title:
Visual Regression Testing for Web Applications

Date: April 10, 2014 **Pages:** vi + 71

Major: Data Communication Software **Code:** T-110

Supervisor: Professor Jukka K. Nurminen

Instructor: Juhani Snellman M.Sc.

Content of the web has transformed from static pages to complex web applications. To ensure software quality, the functionality of an application user interface should be tested with unit and integration tests. However, modifications to the application can result in unintended visual changes, which cannot be detected with functional tests. This visual regression can be tested with a method where the application state is captured as a screenshot and compared with images from previous versions of the application.

The goal of this thesis is to evaluate the existing visual regression testing tools and to find the common problem areas and advantages of current testing techniques. We will also implement a new tool to perform the visual regression testing in Continuous Integration environment.

Our findings show that only two of the four evaluated tools were sufficient for automating the testing. Because these tools had the ability to interact with the tested application and to test individual display elements, they were able to test the entire application. Our evaluation also revealed that the most significant issue in visual regression testing is the difficulty of keeping the visual regression tests up-to-date with the tested application.

We utilized our evaluation findings to design and implement a new visual regression testing tool Giffidiffi. Giffidiffi supports the developers' workflow and provides a user interface to manage the tests. With Giffidiffi, the developer can easily keep the tests updated. Additionally, Giffidiffi will be released as an open source project and the architecture allows to extend and to improve the tool further.

Keywords: visual, regression, testing, Web application, User Interface, Giffidiffi

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mikko Vesikkala		
Työn nimi:	Selainpohjaisten ohjelmistojen visuaalinen regressiotestaus		
Päiväys:	10. huhtikuuta 2014	**Sivumäärä:**	vi + 71
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	FM Juhani Snellman		

Selainpohjaisten ohjelmistojen käyttöliittymien toimintaa voidaan testata yksikkö- ja integraatiotesteillä. Testit eivät kuitenkaan tarkasta käyttöliittymän visuaalista ilmettä, mikä voi johtaa siihen, että sovellusta muokattaessa visuaalinen ilme saattaa muuttua vahingossa. Näitä visuaalisia muutoksia voidaan testata automaattisesti vertailemalla kuvakaappauksia uudesta ja vanhasta versiosta.

Tämän diplomityön tarkoituksena on arvioida olemassa olevia testaustyökaluja sekä löytää visuaalisen regressiotestauksen yleiset ongelma-alueet. Lisäksi luomme uuden työkalun visuaalisen regression testaukseen.

Työssä arvioitiin yhteensä neljää testaustyökalua ja niiden soveltuvuutta visuaalisen regressiotestauksen automatisointiin. Arvioinnin tulokset osoittivat, että koko sovelluksen testaus on mahdollista vain kahdella työkalulla. Lisäksi huomasimme, että testien pitäminen ajan tasalla on nykyisillä työkaluilla vaikeaa. Visuaalisen regressiotestauksen suurin ongelma on nopeasti vanhenevat testit, sillä testien pitää huomioida pienetkin muutokset käyttöliittymässä.

Työn tuloksena kehitimme uuden testaustyökalun, Giffidiffin, jolla ohjelmistokehittäjät voivat helposti testata visuaalista regressiota. Hyödynsimme työkalun suunnittelussa tehdyn vertailun tuloksia ja toteutimme Giffidiffiin selainpohjaisen käyttöliittymän, jonka avulla kehittäjien on helppo pitää testit ajan tasalla. Giffidiffi julkaistaan avoimena lähdekoodina, joten sitä on helppo kehittää tulevaisuudessa hyödyntämään esimerkiksi uusia kuvanvertailutekniikoita.

Asiasanat:	visuaalinen, regressiotestaus, Web-sovellus, käyttöliittymä, Giffidiffi
Kieli:	Englanti

Acknowledgements

In the first introduction lecture in 2006, Jukka Parviainen presented us the original plan of how we should complete our M.Sc. studies by 2011. I did not quite make it in five years but it has been a fabulous journey.

First, I would like to thank my instructor Juhani Snellman and my supervisor Professor Jukka K. Nurminen for helping me through the final steps. I would also like to thank Elisa and Matti Liski for letting me use `kauppa.saunalahti.fi` for this study, and the entire WSP-team for some great Giffidiffi ideas. Additional thanks go out to all my colleagues; you make Reaktor an awesome place to work and it has been a great opportunity to learn from you.

Special thanks go to my friends at the fellowship of an ex-Indian chief as well as to everyone at `#kumikanaultimate`.

Last, but not least, I want to thank my family for all the support, and especially Maiju, you have made this possible.

Espoo, April 10, 2014

Mikko Vesikkala

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CI	Continuous Integration
CSS	Cascading Style Sheets
DART	Daily Automated Regression Tester
DOM	Document Object Model
FSM	Finite State Machine
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
MVC	Model View Controller
NoSQL	Not only SQL
RAM	Random Access Memory
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSD	Solid-state Drive
UI	User Interface
WWW	World Wide Web
XML	Extensible Markup Language

Contents

Abbreviations and Acronyms **iv**

1 Introduction **1**
1.1 Problem statement 2
1.2 Structure of the thesis 3

2 Background **4**
2.1 Web applications 4
2.1.1 Presentation tier 5
2.1.2 Business logic tier 6
2.1.3 Data tier 6
2.1.4 Client-server communication 7
2.2 Software testing 7
2.2.1 Unit testing 7
2.2.2 Integration testing 8
2.2.3 System testing 8
2.2.4 Continuous Integration 9
2.2.5 User interface testing 10
2.3 Web application testing 12
2.3.1 Client-side testing 13
2.3.2 Server-side testing 15
2.3.3 Existing tools 15
2.3.4 Visual testing 17

3 Regression testing **18**
3.1 Regression testing process 19
3.2 User interface regression testing 21
3.3 Web application regression testing 22

4 Evaluation of existing tools **23**
4.1 Evaluation methods 24
4.1.1 Test objectives 24
4.1.2 Case application 25
4.1.3 Test arrangements 28
4.2 Evaluated tools 34
4.2.1 PhantomCSS 34
4.2.2 Huxley 35
4.2.3 Needle 36
4.2.4 Depicted 37
4.3 Evaluation 38
4.3.1 PhantomCSS 38
4.3.2 Huxley 40
4.3.3 Needle 42
4.3.4 Depicted 42
4.3.5 Findings 43

5 New tool for visual regression testing **46**
5.1 Design 46
5.2 Architecture 48
5.3 Implementation 51
5.3.1 Server 51
5.3.2 Test runner 54
5.3.3 User interface 56
5.4 Evaluation 57

6 Discussion **60**
6.1 Implications of the results 60
6.2 Review of research methods 62
6.3 Future work 63

7 Conclusions **65**

Bibliography **66**

Chapter 1

Introduction

Business, social interaction, and everyday life continuously shift their focus towards the Internet and World Wide Web. Web content has transformed from static pages to web applications, software that runs inside a web browser.

The number of web applications is growing fast. The rapid evolution of web browsers and technologies has enabled the building of more advanced applications. Applications vary from internal corporate process tools to small smartphone applications and extremely popular applications, such as the social networking service Facebook.

Application User Interfaces (UI) are built using standard web technologies. HyperText Markup Language (HTML) describes content, Cascading Style Sheets (CSS) defines user interface layout, and JavaScript creates user interface functionality. Because web applications utilize standardized technologies, they are platform independent and accessible with various devices, such as laptops or mobile phones, regardless of the operating system or hardware architecture used [48]. With the new HTML5 specification [7], web applications are becoming increasingly expressive and comparable with old-fashioned desktop applications.

Web applications, as all software, should be tested thoroughly to ensure system quality and to prevent unexpected behaviour. Application user interface can be tested with various tools and methods for proper display of data and components. Since the user interface is in a significant role in web applications, it should be given sufficient attention to ensure its quality and operation. However, visual aspects of the user interface are difficult to test.

Developing a new feature or fixing a software bug can result in unintended visual changes in those parts of the user interface that are already reviewed and accepted. For example, a change in software code or interface styles can move a display element, change font colour, or even break the whole layout.

These changes can be unintentionally ignored if the application developer does not manually review each view and functionality for visual complications after the changes.

To prevent this type of degradation, we should test the software for regression issues. Leung and White [32] define regression testing as a process of testing the modified program to ensure the correct behaviour. Regression testing is applied whenever the software or specification is modified, thus it aims to prevent degradation of the quality of the software.

With agile software development methods, such as Scrum or Lean Kanban, the problem becomes even more significant. The specifications change during the development cycle and the software is constantly revised, enforcing changes also in the user interface.

1.1 Problem statement

Visual regression testing means the approach of testing the UI for unintended visual changes. In this study, we focus on testing the visual regression with a method where the application state is captured as a screenshot and compared with images from previous versions of the software. For each test, one of the previous screenshots is selected to be the comparison image. This image is referred to as the *reference* or *baseline* image.

Recently, several testing tools, such as PhantomCSS [16], Huxley [8] and Needle [14], have tried to address the visual change issues by performing visual regression testing with the forementioned "capture and compare" - method. In an ideal situation, different views of an application would be automatically tested for visual regression. This would take place in a Continuous Integration (CI) environment requiring minimum amount of human interaction. However, user interface tests become outdated easily when the tested software is modified [38, 40]. Because of this, visual regression testing is difficult to automate and the application user interface will in practice always require a thorough manual review. For a complex application with multiple views and functionalities this can be extremely time consuming and leave some defects unnoticed.

The goal of this thesis is to examine the existing visual regression testing tools and methods. Additionally, we will implement a new tool for performing the visual regression testing in an automated Continuous Integration environment. In this study, we attempt to answer the following research questions:

- What are the advantages of the existing testing tools and how suitable are they for visual regression testing?
- What are the special problem areas in web application visual regression testing?
- How can we combine the advantages of the existing visual regression tools?

This study focuses on standard web technologies and rich web applications, where most of the content is rendered with JavaScript on the browser. Other user interface technologies, such as Adobe Flash[1] and Adobe Flex[2], exist but are left out from the scope of this thesis.

1.2 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 provides necessary background information about software and user interface testing. Furthermore, we also review the current state of art in web application testing.

Chapter 3 focuses on introducing the basics of regression testing, the techniques and processes used. We will inspect user interface regression testing and web applications in detail.

Next, we examine and evaluate the existing tools in Chapter 4 by testing them in practice with predefined test scenarios. The test setup consists of an e-commerce application and ten different test scenarios, each with a different visual problem that has been ignored by other tests. The test arrangements and measurements are described in Section 4.1.

In Chapter 5, we introduce a new testing tool based on the findings from the literature review and evaluation of existing tools. Actual implementation is discussed and the new tool is evaluated.

In Chapter 6, we discuss on the topic and share some thoughts about the future work. Finally, we conclude the thesis in Chapter 7 with a summary on the work done.

[1] `http://www.adobe.com/products/flashplayer.html`
[2] `http://www.adobe.com/products/flex.html`

Chapter 2

Background

In this chapter, we will discuss the background of web applications and software testing to gain understanding for the visual regression problem. First, we examine web application architecture and review common technologies. Next, we examine the literature on software testing methods and introduce the common user interface testing methods and problems. Finally, we introduce the web application testing methods in Section 2.3 and review the common testing tools.

2.1 Web applications

When software utilizes web browser and its capabilities on the client side, the client it is commonly referred to as a web application. Jazayeri [30] lists a number of modern web applications, such as Google[1], Wikipedia[2] and Flickr[3] as an example.

A typical web application has a three-tier architecture (Figure 2.1) where the data, business and presentation logic are separated. Data logic provides access to application data stored to the server. In modern applications, the business logic is partitioned to both the client and the server, which communicate over HTTP. Presentation tier consists of user interface components and user interaction.[45]

In addition to the three-tier architecture, web applications can be described with multiple design patterns depending on their implementation. For example, Leff and Rayfield [31] discuss on the usage of Model-View-Controller (MVC) design pattern in web applications. In their approach, the

[1] `http://www.google.com/`
[2] `http://www.wikipedia.org/`
[3] `http://www.flickr.com/`

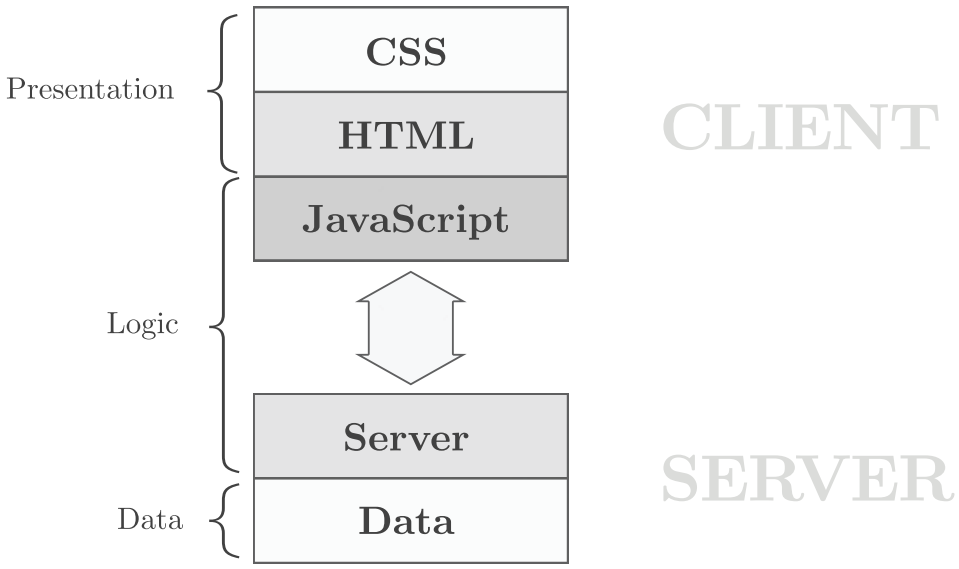


Figure 2.1: Three-tier web application architecture, adapted from [45]

application server employs Models and Controllers to generate Views that are then displayed on the web browser. Jazayeri [30] discusses on how well the MVC approach coheres with web application development and presents a number of modern frameworks that practice MVC design pattern. Typically, Models represents the data store and Controller implement the business logic, eventually creating HTML documents for the View.

However, reflecting our research problem, we are rather interested in the overall application architecture and underlying technologies than specific client or server design patterns. Therefore we will review the common components of web applications: presentation, business logic, data and client-server communication.

2.1.1 Presentation tier

The presentation layer on web application user interface is built with HTML and CSS. HTML documents consist of elements, such as headings, tables or lists [6]. Elements can have textual value as well as additional attributes, such as "id" or "class". The browser receives an HTML document from the server, interprets the content and builds a Document Object Model (DOM). DOM is a structural tree representing the current content and structure of

the web page.[5]

Cascading Style Sheets (CSS) is a style sheet language for structured HTML documents. With CSS, developer can define various aspects for elements, such as the size, margin, color and background image [1]. Together with HTML, CSS defines the web page or web application layout. Eventually, the browser renders the content using the DOM and CSS styles.

2.1.2 Business logic tier

Client-side application logic is implemented with JavaScript. JavaScript is an object-oriented programming language implementing ECMAScript standard [13]. JavaScript is supported by all modern browsers, thus it has an important attribute of being a cross-platform programming language [13]. The browser executes JavaScript to implement client-side logic by fetching data from external resources, modifying the DOM and listening to the DOM events, such as mouse clicks or text field inputs. This allows the creation of dynamic web pages and complex web applications.

Paulson [44] specifies the introduction and adaption of Asynchronous JavaScript and XML (AJAX) as an important breakthrough for the web and web applications. AJAX enables web pages to asynchronously retrieve resources from the server without reloading the whole content. With the help of AJAX, it is possible to create single-page web applications, where all client-server communication is done asynchronously and application state changes do not require a full-page retrieval from the server [41].

Server-side application logic can be implemented with a wide range of different languages and frameworks. The application server can serve the content in two ways: as server-side rendered HTML content, or as data via a RESTful interface. The pre-rendered HTML is displayed directly on the client while the plain data requires client-side JavaScript logic for presentation.

2.1.3 Data tier

As the web technologies, namely HTML, CSS and JavaScript, do not affect the server side components, web applications usually rely on generic databases and store data in SQL or NoSQL databases on the server [45]. In addition to server side storage, modern browsers can store structured data to Web Storage [19] and key-value data to Indexed Database [10]. Web applications can use client side storage to cache data or to allow offline usage.

2.1.4 Client-server communication

Client and the server communicate using the Hypertext Transfer Protocol (HTTP) [9] to share HTML, JSON, images or other resources over the Internet. Web applications typically employ some architecture or protocol built on top of HTTP, such as SOAP[4] or REST, to transfer application data between the client and the server.

2.2 Software testing

Software testing is conducted to determine incorrect software behaviour and to identify failures. Software is executed and the output of a function, module or system is compared with its specification by an external oracle [43, 50]. According to Whittaker [50] the process of actually executing the software code is an essential activity in software testing and distinguishes it from the static code reviews.

To understand the field of web application testing we have to examine software testing methods on a general level. Next, we will briefly review the main activities in software testing.

2.2.1 Unit testing

Myers et al. [43] define unit testing as a process of testing the building blocks of a program, such as functions, classes, routines, or subprograms. Testing individual software components rather than complete modules or the entire system at once has several benefits. Unit testing allows the software developer to focus on smaller and more manageable units. This restricts the problem domain and helps to debug failures and incorrect behaviour. Moreover, unit testing can be conducted in parallel for different modules, which helps to reduce the time used to run the tests [43].

Unit testing is mostly white box oriented testing where the component's internal structure and data flow are known to the developer, thus allowing the development of detailed test cases [43]. The input domain is defined for each unit based on unit specific scenarios and specifications. When each unit is tested, the rest of the system is ignored. Consequently, the unit testing environment may have to provide required software modules. This is achieved by constructing throwaway stubs and drivers offering fixed input data to the unit under test [50].

[4] `http://www.w3.org/TR/soap/`

Runeson [47] emphasizes the importance of unit test automation and how the developers and development teams should focus on automating the test case execution and result checking. Repeatable and automated unit tests executed by the build system provide fast feedback and visibility on the development process. Additionally, automated tests also continuously watch for regression problems.

Unit testing assures the system functionality and is the foundation of the testing procedure [43, 47]. However, higher-order testing, such as integration and system testing, must be conducted to ensure system quality [43].

2.2.2 Integration testing

Integration testing is conducted after the functionality of individual software components has been tested with unit testing. All the units and components are combined into modules and eventually into a working software and the result is then tested by performing integration tests [50]. The main goal for integration testing is to find errors that are left undetected during the unit testing [33].

Testing focuses on the communication between modules and ensures that the interfaces function as specified [50]. Leung and White [33] define three objectives for integration testing. First, the tests must ensure that all module functionalities are provided as specified. Next, the communication interfaces are checked for correct arguments and correct input. Finally, the software is checked for unexpected behaviour and exceptions created by incorrectly accessed functions or fields.

Integration testing can be approached in various ways. Myers et al. [43] list six different testing strategies for integration tests: bottom-up, top-down, sandwich, modified top-down, modified sandwich and big-bang testing. However, these testing strategies do not yield any actual test generation guidelines [33].

2.2.3 System testing

After integration testing, the software should be tested with system level tests. Myers et al. [43] define system testing as a process of comparing the system with its original objectives. They emphasize the importance of testing the system in its real environment, since the testing process should not be limited to the software itself but rather test the entire domain. Additionally, system testing is reasonable only when the system has written and measurable objectives.

2.2.4 Continuous Integration

In agile software development methods, the development is done in small iterations in order to effectively respond and adapt to change [35]. From the software testing perspective, the respond to change is to integrate the testing process into the development.

Continuous Integration is a software development practice aiming to improve software quality by reducing integration problems. Originally emerging from agile development methodology Extreme Programming, CI is nowadays often utilized alongside agile development methods and thus widely adopted by software development teams. [27]

Duvall et al. [26] resolve Continuous Integration to a simple practice where all developers commit only properly tested code to the version control repository. By running private builds the developers ensure the integrity of the software before committing the code to version control. Fowler and Foemmel [27] describe the practice as a continuous process:

- Make required changes to the software code and tests.
- Run an automated build on the developer's workstation. The build compiles the code and runs automated tests on the software.
- Fix broken tests, if any.
- Commit changes to the version control repository.
- Run an automated build on an integration machine.
- Ensure that the build is successful.

According to Fowler and Foemmel [27] Continuous Integration helps to find and remove software defects and integration problems. By executing automated self-testing builds on a Continuous Integration server, the developers have a great visibility on the status of the project. Failing builds and software bugs can be fixed early on the development process rather than waiting for the long integration phase at the later stages of the development cycle.

Continuous Integration focuses on automating builds and test executions. Version control repository plays an important part in automating the build. The development team should store in the repository everything needed for building the software. This way everyone can check out the code from the repository and build the software without the need to search for scattered source files or dependencies. However, the build results should not be stored

in the version control repository as builds are easily recreated by fetching the previous version from the repository and building it from scratch.[27]

Self-testing builds should be automated and executed after each change made to the repository. Fast builds provide fast feedback for the developers and help to fix bugs and integration problems quickly after the change [27]. Unit and integration tests should be run for each build while system testing may be conducted periodically [26]. However, according to Myers et al. [43] system testing is reasonable only when the system has written and measurable objectives. These objectives may not always be available when using agile development methods, therefore we will focus on the unit and integration testing levels.

2.2.5 User interface testing

User interfaces, or Graphical User Interfaces (GUI), are generally considered difficult to test [38, 47, 51]. Graphical user interfaces consist of various graphical elements, such as text boxes, buttons and images. User interacts with the UI for example by typing text or clicking an element using a mouse or other pointer device, such as a touchscreen. The software then interprets the interaction *event* and possibly changes the software *state*. A common UI testing technique is to record these events and then repeat them to verify the UI state [38].

Various testing tools are used to capture and record the mouse and keyboard events that developers generate by interacting with the UI [38]. Recorded events can then be repeated to verify the final state of the interface. A test oracle is used to evaluate if the UI appears to be in the correct state, usually comparing with the state captured during the initial test case record phase [38].

Several techniques for UI testing, test case design and test case generation have been presented [51]. However, most of the testing techniques researched focus on only one aspect of UI testing, such as test-case generation, therefore failing to cover the UI testing problem as a whole. Memon [38] also describes the record-playback UI testing techniques as "incomplete, ad hoc, and largely manual". Eventually, this has led to a situation where none of these methods has been widely adopted by the software industry.[39]

User interface testing has unique problems, such as the great amount of manual work and ad hoc testing. Memon [38] offers five basic steps as a general solution to the UI testing problem:

1. Define coverage criteria and specify test scope by selecting the views to be tested.

2. Specify mouse clicks and text field inputs using the software specification.

3. Specify expected output such as element positions, text content or screenshots.

4. Execute test cases and compare the final UI state to the expected output.

5. Analyse test results and determine if the UI was properly tested.

UI test cases are manually crafted and handed out to a test execution engine, which runs the tests and provides aggregated results on how well the tests executed. The automation of this type of testing appears to be a major problem [47]. Automation can save resources and provide a great speedup for the testing and development processes. Despite the automation support of typical capture and replay tools, developers must still analyse requirements, generate test cases, run tests and maintain test scripts when changes are made to the software [51]. With constantly changing software, the tests become outdated easily and require extra work from the developers.

However, only few UI test automation solutions have been presented. Xiaochun et al. [51] present a solution that consists of a test case management dashboard, a test driver, an execution engine and a reporting functionality. First, the test cases are designed via a web-based dashboard tool using a set of predefined keywords to describe the cases. After this, the test cases are interpreted by a test driver. The driver parses the test descriptions, maps the actions to the correct UI elements and hands the complete test scripts to an execution engine. Finally, the execution engine runs the scripts automatically and records the output to a report that summarizes the results. This automation solution supports the entire test procedure from test case design to test reporting. However, the solution has not gained industry support as it lacks a proper testing tool implementation and the test execution depends entirely on the underlying UI technology.

Chang et al. [23] have a different approach as they present a new Sikuli test tool for automating the UI testing. The Sikuli Test employs computer vision to test the visibility and correctness of UI elements in automated tests. First, the developer converts the test cases to visual test scripts consisting of action and assertion statements (Figure 2.2). Sikuli Test is then used to execute the test script and finally to determine whether the test was successful or unsuccessful. The test cases are easy to read and allow non-technical people to write the cases. Furthermore, Sikuli Test is also platform independent solution as it can run on any GUI, including web applications.

Figure 2.2: Sikuli test script example [23]. The *click* action statement and *assertExist* assertion statement take images as arguments. This example script clicks on a color palette and assumes that a color picker becomes visible.

2.3 Web application testing

To understand the mechanics of web application testing and the original problem of visual regression, we need to review the basics of web application testing. We focus mainly on the client-side where the user interface is eventually displayed on screen.

Traditional software testing approaches, such as test models and processes, are applicable also to web applications [25]. However, in contrast to traditional desktop software, web applications have many special problems that make the testing challenging. Myers et al. [43] argue that a large user base, the challenges of the business environment, user's high expectations and browser compatibility issues make web applications difficult to test. Bugs and defects in the application will have quality conscious customers switch to competing services.

The three tiers of web applications - data, business logic and presentation - each have their own problems and challenges. Myers et al. [43] suggest that each tier should be tested independently with separate areas of interests to match the tiers' challenges. However, it might be difficult to separate tiers to coherent and testable components, since in modern web applications the business logic is often distributed between the server and the client.

Respectively Hieatt and Mee [28] propose a three-step process to test web applications. First, parts of the server-side code that are not directly involved in rendering the HTML are tested. Next tested parts are the parts of client-side code that require no interaction with the server. Such components can be field validations or display of modal views. Finally, the client-server communication is simulated and the output of the server software tested. The process proposed is, however, mainly suitable for server generated pages, as the server is expected to return ready-made HTML content. This is not the case for dynamic AJAX web applications that render most of the content on the browser using JavaScript.

Di Lucca et al. [24] propose a typical two-phased unit and integration testing process to conduct the functional testing of a web application. First, unit tests are used to test the functionality of both server and client side pages. This requires lightweight stubs as well as a proper driver for the tests, such as a capture and replay tool. Finally, after the pages are tested separately, use cases are converted into integration test sequences and that are executed on the application views. Test results are verified using invariants to assert the application state.

2.3.1 Client-side testing

Di Lucca et al. [25] divides web application testing models into two categories: *structural* and *behaviour testing models*. Structural testing models are derived directly from the web application implementation by crawling the web application and extracting the state information. Behaviour testing models describe how the application is expected to function, without any knowledge of the actual implementation.

2.3.1.1 Structural testing

A semantic sequence based web application testing has been proposed by Marchetto et al. [34]. Their approach generates semantic interaction sequences from a state model extracted from the application with dynamic code analysis. Test sequences consist of various interaction events that cause the desired state changes. Tests are executed with a record and replay test tool Selenium, which replays the events and checks output values from DOM, thus it verifies the consistency of the sequence.

A somewhat similar approach is presented by Mesbah and van Deursen [42], whose method for the automatic testing of AJAX web applications employs dynamic analysis and to construct a state-flow graph of the application. Their testing tool then automatically fills in forms and click elements to crawl through the states in the state-flow graph. The application DOM is simultaneously tested for faults and errors.

According to Artzi et al. [22], the dynamic analysis, which is used to generate state-charts, is unable to detect all event handlers. Therefore the generated state-chart is incomplete and will fail to test the application thoroughly. As a solution, Artzi et al. [22] present a framework for automated JavaScript web application test generation. The framework employs feedback-directed random testing where every set of generated tests is guided by the feedback of previous test executions, hence improving the test coverage continuously.

All of the testing techniques introduced are somehow based on crawling through the application and extracting the state information out. This type of crawling and the structural approach has two major problems when used for rich JavaScript applications [25]. First, the application can have numerous different states, therefore causing the number of possible state combinations to grow exponentially. Second, some of the states can be very dynamic and thus difficult to reproduce under certain conditions.

2.3.1.2 Behaviour testing

Di Lucca et al. [24] present a black box testing approach in which the web application is tested by using decision tables. In their technique, the application state and its behaviour is expressed with decision tables. The table (Table 2.1) consist of fields describing the *input variables*, *input actions*, *expected results*, *expected output actions* and the application *state before the test* as well as *expected state after the test*. Each variant describes a functionality and the expected behaviour derived from the requirements and use cases. These tables are then used to create the test cases and required stubs and test drivers, which the authors describe as a web page that interacts with the application under test by filling in forms and clicking buttons and links.

Variant	Input section			Output section		
	Input variables	Input actions	State before test	Expected results	Expected output actions	Expected state after test
...	...			...		

Table 2.1: Decision table example [24]

Another approach is proposed by Andrews et al. [21], who model web application behaviour with a Finite State Machine (FSM) technique. At first, the application model is formed from a hierarchical set of state machines. Lower-level state machines modelling individual web pages are aggregated eventually into a single application-wide finite state machine. Resulting FSM is used to generate the test cases, consisting of transition sequences and constraints, in the way that all nodes and all edges of the FSM graph are explored. The web page FSM modelling and model aggregation may be conducted by an automated crawler, while some stages of the approach remain solely manual and require input on expected behaviour.

2.3.2 Server-side testing

Traditional testing techniques and testing tools can usually be employed when testing server-side code [25]. As proposed by Hieatt and Mee [28], those server-side components that do not directly construct any HTML, should be tested first.

In rich JavaScript applications, most of the rendering takes place on the client and only small amounts of the content is pre-rendered in the server. Usually these server-side components are encapsulated in interfaces and exposed as web resources. These interfaces can be tested with standard assertion and invariant techniques easily, thus the level of independently testable server-side code is fairly high even in integration test level.

2.3.3 Existing tools

Test tools and frameworks have an important role also in web application testing. Server-side code can be tested with traditional testing techniques and tools but the client-side application requires different testing utilities.

Mocha [12] is a widely used test framework used to test JavaScript code. In addition, it allows the developers to write client-side functionality tests in JavaScript with support for different browsers. Mocha can be used to test the web application in two ways. Firstly, Mocha can be used to write unit tests for the client-side code. Secondly, the developers can utilize Mocha to test the actual UI functionality.

The UI testing is conducted by writing asynchronous test code that simulates user actions, such as button clicks and text inputs, and verifies the application state from the DOM or global JavaScript state afterwards. The test script is then executed over the application within a browser session. An example Mocha UI test script is presented in Listings 2.1.

Jasmine [11] is an alternative JavaScript test framework which can be utilized to write unit tests for JavaScript code and execute web application UI tests. Jasmine tests are highly similar to Mocha tests and are constrained by the same limitations. Jasmine tests are also unable to verify any visual aspects of an UI element other than basic visibility or CSS properties.

```
describe("Products", function() {
  var products = Products()

  describe("Shopping", function() {
    before(products.loadPage)

    describe('Basic rendering', function () {
      it('Shows correct prices on page load', function() {
        expect(ballProduct().find('.price').text())
          .to.equal('33,13')
        expect(ballProduct().find('.monthly').text())
          .to.equal('10,00 e / kk')
        expect(ballProduct().find('.single').text())
          .to.equal('1,20 e')
      })

      function ballProduct() {
        return products.productElement('Ball')
      }
    })
  })
})
```

Listing 2.1: Example Mocha test code

Selenium [18], or Selenium WebDriver, is another popular test tool for conducting web application UI testing. While Jasmine and Mocha simulate user actions by generating the events with JavaScript, Selenium controls the browser via a WebDriver interface and instructs it to generate clicks and other events on the web page.

WebDriver [20] is an interface for controlling the behaviour of a web browser. Using the WebDriver interface, Selenium can instruct the browser to navigate to a web page and conduct various actions, such as clicks and text inputs. Furthermore, WebDriver interface also allows the test to read the DOM state and evaluate arbitrary JavaScript on the target page. Thus WebDriver can be used to automate UI tests by programming the events and verifying the result from the DOM. WebDriver specification is supported by all major browsers, including Google Chrome, Mozilla Firefox, and Internet Explorer.

Mocha and Jasmine can be used to test and verify that the user interface functions as expected when, for example, user clicks a link or submits a form. Even though the tools can verify the visibility and various CSS properties of an individual element, they cannot verify the actual rendered appearance. Selenium WebDriver can be utilized to take screenshots of the web page or individual element on the page but the tool itself does not support any methods for conducting visual regression testing. Two of the existing layout regression testing tools evaluated in Section 4.2, Huxley and Needle, utilize

Selenium to conduct the testing.

2.3.4 Visual testing

Web browsers are capable of rendering complex visual appearances by using HTML and CSS to define the layout. Research on the field of testing these visual aspects of web applications is practically non-existent. Similarly the UI testing in general focuses solely on the UI functionality, disregarding its aesthetic functions.

As discussed previously, the functional testing tools do not take the visual appearance into account and reveal visual problems. The forementioned Sikuli test tool can be used to test web application UI functionality with visual properties [23]. However, Sikuli utilizes visual aspects only to identify pre-defined elements on screen, such as buttons and labels, but does not attempt to detect possible changes in other parts of the UI. Similarly some tools, such as CSSCritic [3], exist to test changes in the CSS definitions but not the actual changes in rendered views.

As discussed in the problem statement (see Chapter 1.1), utilizing screenshots is a common method for testing the visual appearance. The application state is captured as a screenshot and compared with images from previous versions of the software. For each test, one of the previous images is selected to be the reference image. The existing visual regression testing tools compare the image pixel values to determine whether the images look alike or not [4, 8, 14, 16]. This screenshot technique is a simple approach to test the visual appearance but also highly prone to errors because even small changes to the application UI will render the reference images unusable.

Chapter 3

Regression testing

As discussed in Chapter 1, Leung and White [32] define regression testing as a process of testing the modified program to ensure the correct behaviour. Following the definition, they describe the three goals of regression testing:

- Ensure correct logic in modified parts of the software.
- Ensure the functionality of unmodified parts of the software.
- Validate the functionality of the complete modified software.

Runeson [47] emphasizes the important role of continuous regression tests during software the development. Tests should be conducted after every modification to the software, thus reaching the first two goals. The added, deleted or re-written logic is tested together with the unmodified code. Regression tests help to detect incorrect behaviour whenever the software environment changes, new features are added in production, or the production software is otherwise modified.

Leung and White [32] divide the regression tests into two categories: *corrective regression testing* and *progressive regression testing*. In corrective regression testing the software specification remains unchanged while the software itself will have minor modifications to the software code. This type of corrective testing is usually conducted during development cycle or when a program failure is detected and fixed. Progressive regression testing is invoked when a major code modification is introduced and the software specification is changed accordingly. Such change can for instance be adding a new feature or functionality to the software.

Regardless of the regression test type, previously written test cases should be reused as much as possible. As the specification does not change in corrective regression testing, existing test cases are usually still valid. However,

in progressive regression testing the change in specifications and code can require new test cases and changes in the existing tests. Generally only the affected parts of the software need to be retested, yet it can be difficult to determine which parts remain unaffected and can be left out of the testing scope.[32]

3.1 Regression testing process

A typical regression testing process may vary depending on the organization, development team, and software in question. Holopainen [29] identifies seven steps in regression testing:

1. If software component K is tested for the first time, corresponding tests T must be developed.
2. K is tested against the test cases T. If the test cases fail, proceed to state 3. If the test cases are successful, go to state 7.
3. Locate and fix defects from component K creating a changed version K'.
4. Select regression tests T' from the original tests T. New tests may be created and added to both T and T'.
5. Test component K' for regression tests T'. If the test cases fail, proceed to state 6. If the test cases are successful, go to state 7.
6. Fix defects from component K' and go to state 4.
7. Testing for component K has ended.

Continuous regression testing is recognized as a strength, as previously stated, and should be considered a default practice. The process described by Holopainen [29] does not consider continuous regression testing in any specific way but will fit the continuous process when starting from the state *2* with existing tests T. However, selecting separate regression test cases T' from the original tests is difficult and will cause overhead to the regression testing procedure [47]. By simplifying the previously introduced process and leveraging the Continuous Integration practice from Section 2.2.4 we can formalize the process for continuous regression testing and use it as a basis for our upcoming work:

1. Detect a change in the version control repository, caused by a new feature, bug fix or any other code or resource change.

2. Run automated build on the software code C and test the code using test set T, containing all unit tests and integration tests. If any build or test failures are detected, declare the build as failed. If no failures are detected, the build is successful and tested for regression.

3. If a build failure has been detected, locate and fix the defects in code C by creating a change set C'. Implement new or modify existing tests T if needed, creating a change set T'.

4. Commit C and T with change sets C' and T' respectively to the version control repository and return to the beginning of the continuous process.

The process is presented in Figure 3.1.

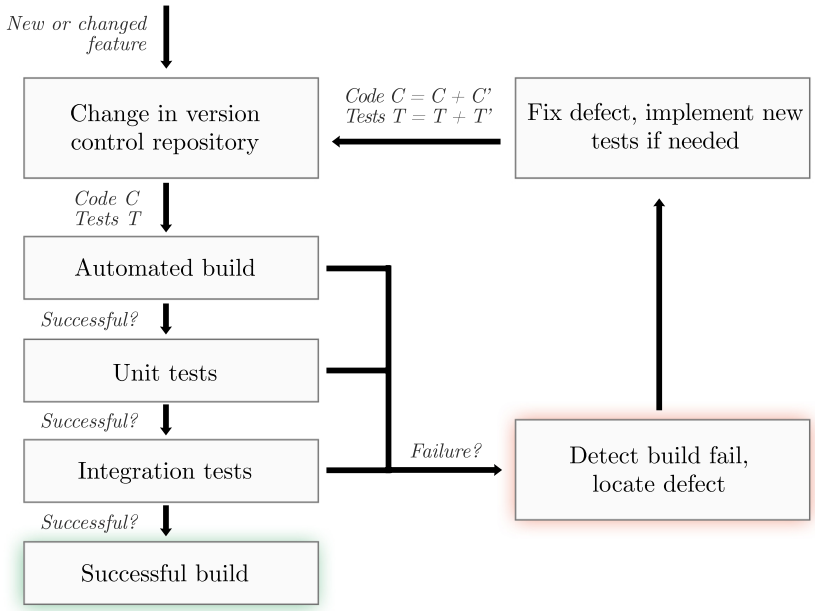


Figure 3.1: Continuous regression testing process

Hence the Continuous Integration practice enforces regression testing when run with the complete test suite. This approach is suitable for both the corrective and progressive regression testing by continuously reusing existing test cases and building on thoroughly tested software. The automated build process, triggered by changes in the version control repository, provides excellent visibility for the development process and helps to detect regression issues shortly after the changes are made [27].

3.2 User interface regression testing

As discussed in Section 2.2.5, UI testing is generally based on interacting with the UI elements and comparing the results with expected output. The approach is extremely prone to errors as even minimal changes in the application layout may render the given input or expected output obsolete if an element for example changes place [38]. Changes in the software would also require changes to the test cases, thus rendering the existing test cases outdated [38].

The research activity on UI regression testing has been fairly low and only a few examples can be found. Memon and Soffa [40] present a regression test model which aims to prevent test cases from degrading and becoming outdated. The model consists of states, objects, events, and invocations which are used to describe test cases. When the UI has been modified, the test cases can be categorized to be either usable or unusable. All unusable tests are run through a repairer algorithm. The algorithm tries to repair illegal event sequences in the unusable test cases so that they can be executed correctly on the modified UI.

Memon and Soffa [40] also present a case study on the regression testing model described above and apply it to two software applications, Adobe Acrobat Reader and their own implementation of Microsoft WordPad. However, the approach has low applicability to industry use as the model lacks an actual testing tool implementation. The process of generating the test cases manually is very time-consuming, especially when the software UI undergoes frequent modifications.

To provide a more practical solution, Memon et al. [36] published DART (Daily Automated Regression Tester) framework for the regression testing of user interfaces. The framework addresses the issue of frequent re-testing of software builds by trying to automate the testing process. First DART analyses the UI and saves the structure to a XML document. Then it creates an event-flow graph from the stored structure and allows a software developer to select desired test cases using the graph. Based on those selections, DART generates automatic test cases together with expected outputs. After the software is modified, DART runs it as scheduled and executes all test cases automatically. The tests are appraised as successful or unsuccessful based on the comparison of actual and expected output. However, DART is very reliant on the UI technology used, which causes it to be tightly coupled to the underlying software platform [36]. Ripping a Java application UI poses different requirements than analysing a DOM and CSS of a web application, for example. The testing conducted is also mainly functional testing, thus it does not test any visual properties of UI elements. Because of this, DART

does not solve the visual regression testing problem.

3.3 Web application regression testing

Web application regression testing research has mainly discussed the creation of separate regression tests by using methods of test case generation [46, 49]. However, as discussed in Section 3.1, the continuous regression testing process ensures that the regression problems are caught during the development. Thus extensive unit and integration tests are sufficient and separate regression tests are not needed.

This regression testing process suits web applications, even though they have a three-tier architecture and pose a few special challenges for testing (see Section 2.3). web applications can be tested with typical two-phased testing using separate unit and integration tests [24]. Server-side testing can be conducted with traditional testing techniques, as discussed in Section 2.3.2, and the client-side functionality can be tested with for example the tools introduced in Section 2.3.3. Thus, in the context of Continuous Integration practice, web applications are considered similar to standard software and can be tested in a CI environment. This ensures that the application will also be tested for regression issues (see Section 3.1).

However, as discussed in the introduction (see Chapter 1), the application UI can suffer from unintentional regression issues when the application is modified. Because the UI functionality tests do not reveal these visual changes, separate tests are required to prevent any visual regression. Existing visual regression testing methods rely mostly on capturing the UI state on screenshots and comparing them with reference images from previous versions of the application.

Unfortunately, this type of capture and compare testing is vulnerable to frequent UI changes. As explained in Section 2.3.4, the tests become quickly outdated when the tested application is modified. For this reason, it is extremely important to consider test maintainability when creating the tests.

Chapter 4

Evaluation of existing tools

One of the objectives of this study is to find the advantages and disadvantages of the existing tools. In order to accomplish this, we have to first experiment with the tools and find out how they suit our problem of performing visual regression tests on web application layouts. We examine four different test tools and focus on the overall applicability to the issue while collecting observations on the ease of use from the application developer's point of view. In addition to this, we will also try to identify the common problem areas of visual regression testing.

We found total of six tools that were able to test visual regression. All of the tools utilize similar "capture and compare" -methods to perform the testing. However, we selected only four tools for the evaluation: PhantomCSS, Huxley, Needle, and Depicted. The two other tools, CSSCritic[1] and Wraith[2], were left out of the evaluation scope after initial inspection due to lack of support for CI server integration.

The evaluation is conducted by using the tools to test the case web application, a typical single page online store. Evaluation objectives, test arrangements and case application are presented in Section 4.1. Evaluated tools are introduced in Section 4.2 and evaluation findings are presented in Section 4.3.

[1] `https://github.com/cburgmer/csscritic`
[2] `https://github.com/BBC-News/wraith`

4.1 Evaluation methods

To evaluate the layout regression tools, we decided to use them to perform regression testing on a case application. As we found out previously, test suite automation plays an important role in regression testing and CI [27]. We will take this into account and evaluate the tools in a Continuous Integration environment.

This section describes the evaluation methods more in detail by first stating the evaluation objectives and finally introducing the case application and test arrangements, including the fault seeding methods used.

4.1.1 Test objectives

The first goal for the evaluation is to find out whether the tool could be used to test our case application for visual regression or not. To achieve this, we decided to simply seed the application with faults and measure the exact amount of defects found by each tool [37]. From this data, we should be able to draw conclusions on how well they performed the task. Fault seeding methods are explained in Section 4.1.3.

The second goal is to find the advantages and disadvantages of each tool under inspection. We decided to focus on the characteristics of each tool and examine the features supported and possible defects found. We will examine how the tools support the following features related to visual regression testing and CI environment:

Full page testing
: Full page testing is the simplest version of layout testing. The tool is able to navigate to a page, snap a screenshot and compare it with a previous version.

Individual element testing
: Instead of a full page screenshot, the tool is able to define the individual HTML element to be tested. Thus the developer may test the application in small increments.

Page interaction
: The tool supports interaction, such as clicks on buttons and other elements, with the application under test.

Supports multiple browsers
: The tool is able to run tests on at least two different browsers.

Supports headless browsers
: The tool is able to run tests on at least one browser that operates without displaying a real GUI to the user. Headless browsers are for example PhantomJS and SlimerJS.

Test reporting
: The tool provides test reports compatible with CI server.

Supports individual reference image reset
: Developer can reset reference images for each test separately.

Difference images for failing tests
: When the tool detects a failure in layout, it generates a difference image that visualizes the error.

Finally, the third goal is to analyze how do the tools adapt to changes in the tested application and how easily the developer can repair the outdated tests.

In addition to these, we will gather notes on various findings during the experiment, such as the perceived usability issues and potential problem areas detected. Browser compatibility issues are left out from this study.

We will utilize the theory on regression testing (see Chapter 3) as a foundation for our research and employ the regression testing process presented in Section 3.1 and Figure 3.1. We consider the visual regression tests to be a part of the integration test suite. In our experiment, we will assume the following workflow and focus on the final step while assuming that the steps 1-3 are completed in advance:

1. Developer commits changes to the version control repository,
2. CI server builds the application and executes unit tests,
3. After the tests are completed, CI server deploys the modified application to a test server running with mock data,
4. When the server is deployed, CI server executes visual regression tests as part of the integration test suite.

4.1.2 Case application

The selected case application is an online shopping service for the Finnish telecommunications operator Elisa. In the online store, Elisa sells mobile phone and broadband subscriptions, prepaid subscriptions, internet security

services, and devices, such as mobile phones and laptops. The application language is Finnish. For our purposes, it is relevant to examine the single-page client application more closely to understand the objectives and challenges implementing the visual regression testing experiment.

The case application consists of a single-page JavaScript client and various RESTful background services. The application is written in JavaScript employing libraries such as *Bacon.js*, *Underscore*, and *jQuery*. The user interface is created with HTML5 markup and CSS3 styling. The application communicates with the servers by sending and receiving data in JSON format while caching orders and other data in browser's local storage.

As the application utilizes HTML5 capabilities, it supports only modern browsers, such as Mozilla Firefox, Google Chrome, and Internet Explorer versions 9 and greater.

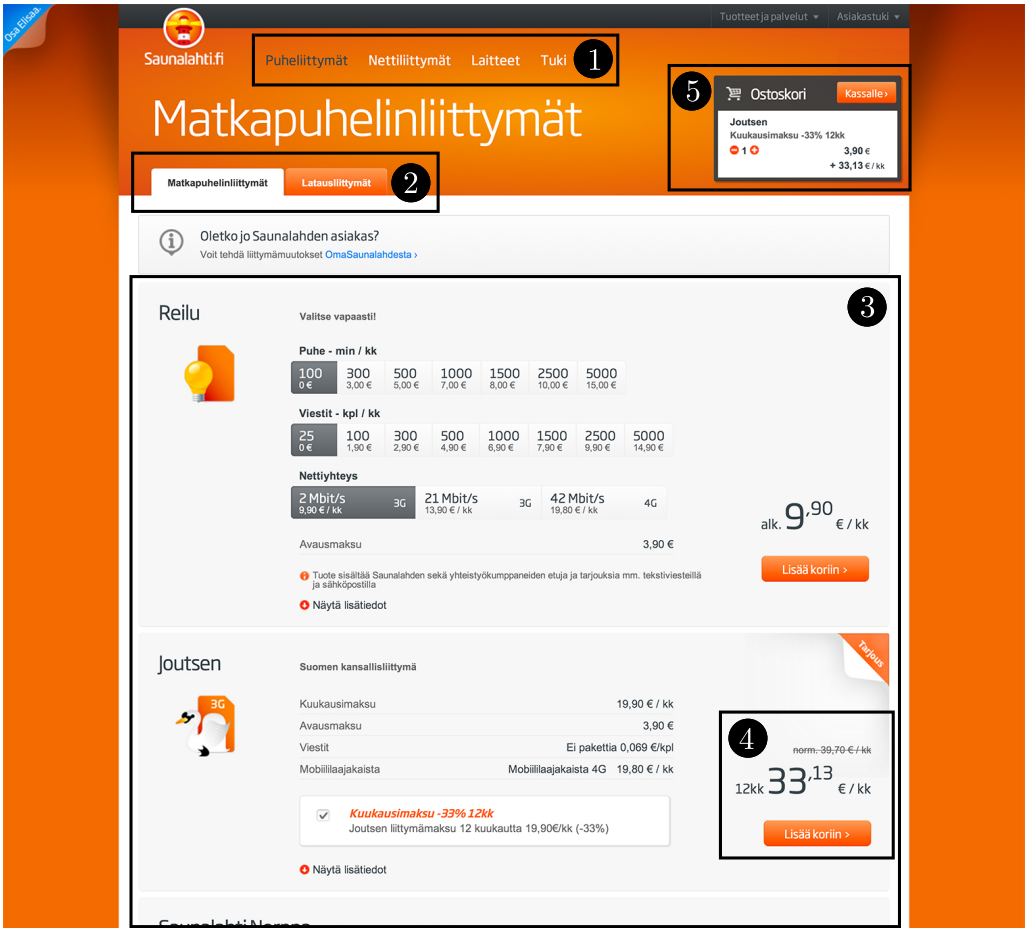


Figure 4.1: Case application: layout overview

The application layout consists of the main navigation, sub navigation, shopping cart, and content areas (Figure 4.1). User can navigate in the application by using four main items from the main navigation (item 1 in Figure 4.1): *Mobile subscriptions*, *Mobile broadbands*, *Devices*, and *Support*. *Support* points to an external site that is not considered a part of the application. Sub navigation (item 2 in Figure 4.1) items change according to each main navigation item selected, for example *Mobile subscriptions* has two sub navigation items: *Mobile phone subscriptions* and *Prepaid subscriptions*.

Products are visible in the content area (item 3 in Figure 4.1). Each item has a name, a visual icon or image, a brief description, and a price element displaying the price and an *Add to cart* -button (item 4 in Figure 4.1). The items may also have a longer description which can be seen by clicking *Show more details* -link. In addition to the basic content layout, the *Device*-pages have additional elements for sorting and searching the device catalog.

When an item is added to the shopping cart, it becomes visible on the shopping cart next to the navigation element (item 5 in Figure 4.1). The user can add multiple items to the shopping cart and adjust the amount of each item on the cart.

The shopping cart checkout is a two-phased process and is displayed as a modal view for the user (Figure 4.2). In the first phase, the user fills in personal details, such as the name and address, and can perform selections regarding to the subscription products in the order. The user can then proceed to the second checkout phase to confirm and verify the order by performing a strong identification through an identification service.

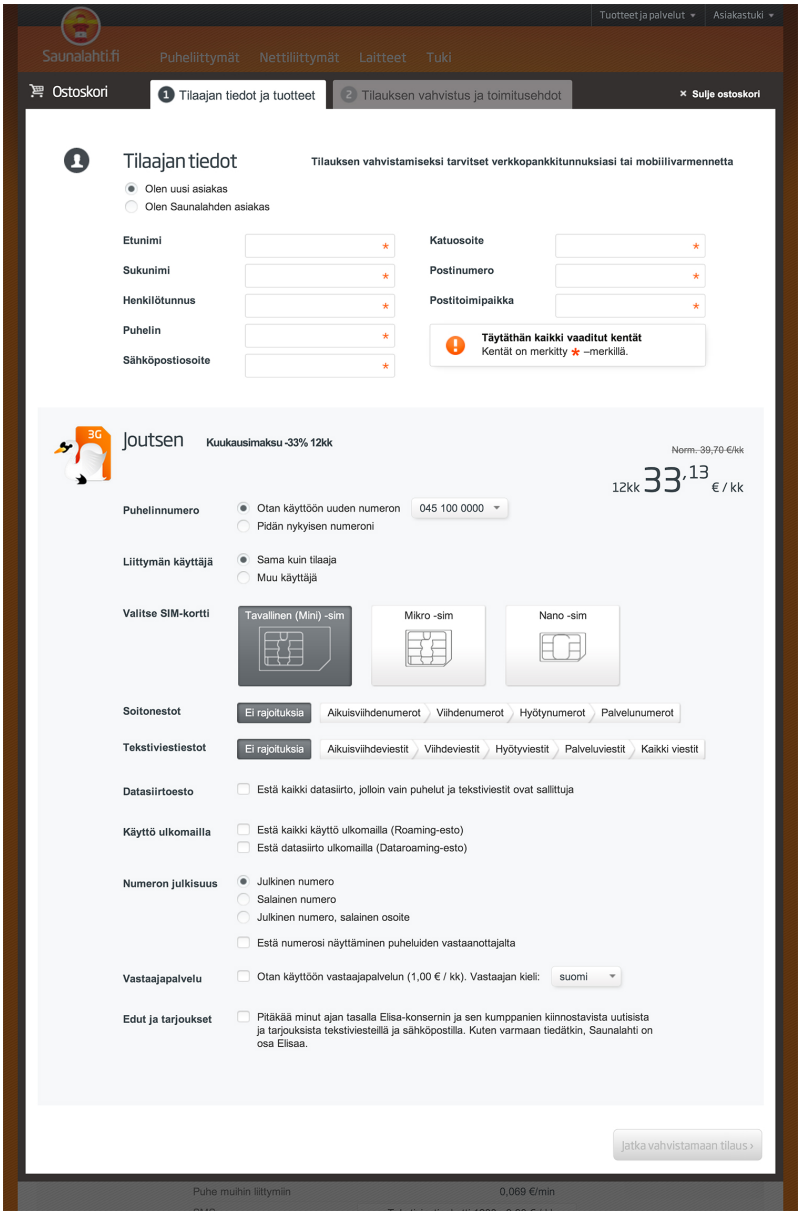


Figure 4.2: Case application: checkout

4.1.3 Test arrangements

Next we will introduce the fault seeding techniques utilized and review the injected faults. Finally, we will define the test execution environment and the tools used.

4.1.3.1 Fault seeding

To be able to test the tools under examination, we need to inject faults to the application. Memon and Xie [37] propose that seedable faults should be categorized according to their behaviour and similar to those made by developers. Furthermore, the faults should be injected into adequately tested code. Memon and Xie [37] also suggest that several faults from each fault category should be injected into the application, still retaining even distribution among categories in order to maintain balanced test setup.

Marchetto et al. [34] employed another interesting technique in their study. They collected real failure description data from the bug tracker, reproduced the errors and injected the errors to the application code before running the tests. Since the faults origin from actual application development history rather than a fault generator, they serve as an adequate representation of real faults and bugs that could occur during the application development.

For our case application, we combined the two fault seeding methods and eventually selected 10 different issues from the development history of the application. The issues were selected to represent different categories of visual defects that could occur during the development and would not be detected by other automated testing. The injected faults are introduced in Table 4.1. As the regression testing conducted by the tools is purely visual, the faults were categorized into two main categories by their effect on the visual representation: *spatial change* and *appearance change*.

Spatial change refers to an unintended change in element's position or sizing that affects the spatial position of the element. Appearance change denotes a change in element's visual appearance, for example a minor change in text color, or a wrong background image. The element size and position remain unaffected. Half of the injected faults were categorized as appearance change (faults from 6 to 10 in Table 4.1).

In addition to the two main categories, the faults were also categorized depending on whether they required interaction with the application or not. Hence, the faults that are visible by simply loading the page are categorized as requiring no interaction whereas other faults require interaction, for example the ones occurring during the checkout process.

Table 4.1: Injected faults. Red color marks the defect.

1 **Category:** Spatial change

Requires interaction: no

Affected views: Mobile subscriptions, prepaid subscriptions, mobile broadbands, security subscriptions

Fault: Remove 7 pixels of margin from Pricing and Add to cart -button widget, therefore moving the elements few pixels right.

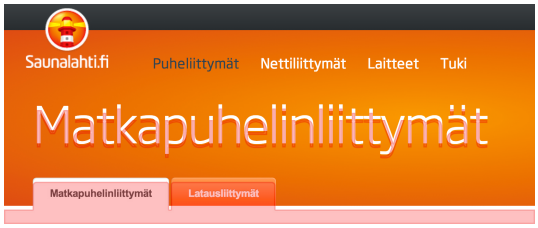
Incorrect placement highlighted in red

2 **Category:** Spatial change

Requires interaction: no

Affected views: All, except checkout

Fault: Add 5 pixels of padding to the page title element, therefore moving the entire page content down.



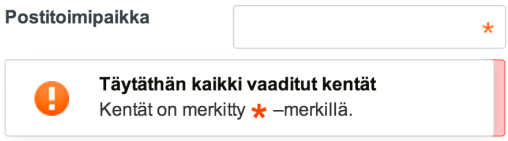
Incorrect placement highlighted in red

3 **Category:** Spatial change

Requires interaction: yes

Affected views: Checkout details

Fault: Add margin to the notification element styles, thus reduce the element width in checkout details view.



Reduced element width highlighted in red

---

4 **Category:** Spatial change

Requires interaction: yes

Affected views: Checkout confirmation

Fault: Reorder DOM elements in checkout confirmation view. This changes the element floating and moves the delivery notification element down.

Incorrect placement highlighted in red

---

5 **Category:** Spatial change

Requires interaction: no

Affected views: All, except checkout

Fault: Increase the font size in Pricing widgets by one pixel.

Incorrect placement highlighted in red

---

6 **Category:** Appearance change

Requires interaction: no

Affected views: Mobile subscriptions

Fault: Remove "campaign" class attribute from the DOM template of mobile subscriptions. This removes the campaign decorator from products that have a campaign.

Correct (*top*) and incorrect (*bottom*) campaign decorator

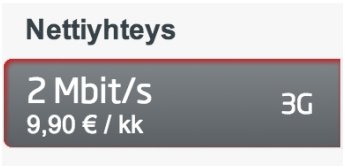
---

7 **Category:** Appearance change

Requires interaction: no

Affected views: Mobile subscriptions, prepaid subscriptions

Fault: Remove box-shadow attribute from button styles, thus affecting also ”Reilu” widget styles.



Missing shadow highlighted in red

---

8 **Category:** Appearance change

Requires interaction: no

Affected views: Devices, Phones

Fault: Change text color to darker grey in device sort widget.

Correct (*left*) and incorrect (*right*) sort element

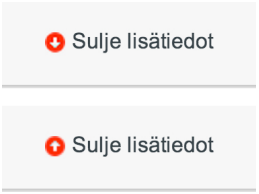
---

9 **Category:** Appearance change

Requires interaction: yes

Affected views: All, except checkout

Fault: Add wrong style to Additional info -button when the button is clicked.



Correct (*top*) and incorrect (*bottom*) button style

---

10 **Category:** Appearance change

Requires interaction: yes

Affected views: Checkout details

Fault: Element order changed in order details, therefore mismatching the CSS selector and displaying wrong SIM card background image.

Correct (*left*) and incorrect (*right*) button style

4.1.3.2 Test execution

For each tool, we created a test suite to test all the content views. Additionally, we created test suites for interactive views, such as the checkout views, for the tools supporting interactivity. To simulate a real use case, the tests were added to Git version control, each test suite under its own repository.

To properly simulate a real CI environment, we built a virtualized Linux installation to act as a CI server. The server is running a commonly used Jenkins[3] CI server with appropriate plugins installed. Furthermore, the server is running a Selenium server for remote-executed WebDriver tests and a Depicted server to run the Depicted test suite. The Depicted server is introduced in Section 4.2.4.

We created a separate Jenkins job for each test suite. The jobs fetched the test suite code from the Git repository, executed the tests with a command line script and parsed the results from either the test program output or xUnit report. The visual regression test suites should be run automatically after the integration tests. However, to avoid any conflicts between different tools in the experiment, the test jobs are run manually after every fault is injected, one at a time.

The application under test is running on a separate server with fixture content. The store content, such as item names and descriptions, is prone to constant change. For this reason, the data is not fetch from the production database but loaded from a fixed data set that is seldom updated. The same fixture data is used when testing the application UI functionality with Mocha tests.

Figure 4.3 illustrates the Jenkins setup with one job for each test suite

[3] `http://jenkins-ci.org/`

and an additional baseline reset job for the PhantomCSS suite.

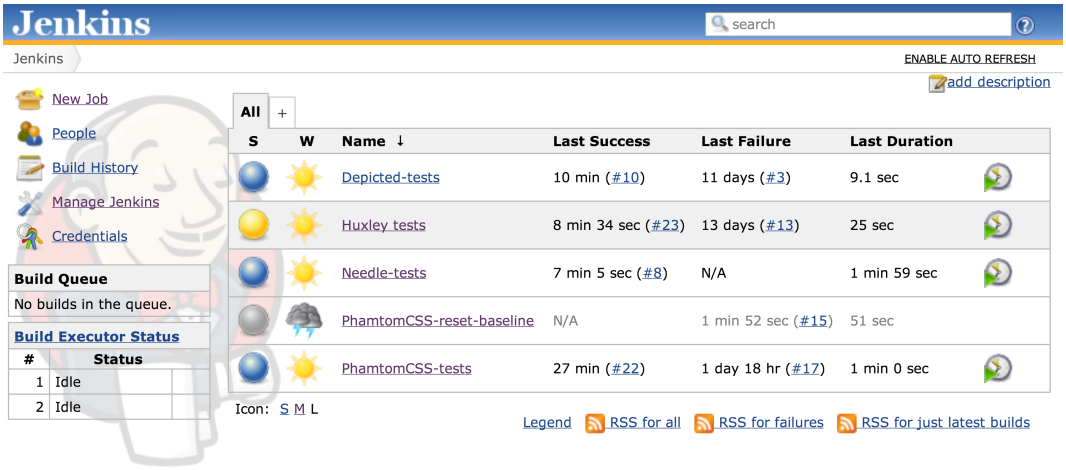


Figure 4.3: Jenkins CI server

The experiment was executed in a virtualized environment running Ubuntu Linux operating system with two processor cores and 8 Gb memory. The host machine is a standard Apple MacBook Pro laptop running a 2.3 GHz Intel i7 processor with four cores, 16 Gb of RAM and an SSD storage. The hardware setup affects only the test executing time and the influence should be fairly low as only one test suite is executed at a time.

4.2 Evaluated tools

Next, we will introduce the evaluated tools and their capabilities.

4.2.1 PhantomCSS

PhamtomCSS [16] is an open source tool for automating visual regression testing. PhantomCSS runs on PhantomJS [17], a headless WebKit browser built for headless web testing and screen capturing. The PhantomJS tests are written in JavaScript. The browser does not support Selenium WebDriver protocol, hence it cannot be substituted with other browsers. PhantomCSS itself is implemented as a CasperJS [2] module, which is a utility library for PhantomJS that provides helper functions for writing the tests.

The tests are written in JavaScript by first using CasperJS to navigate through the application and then PhantomCSS to capture the screen. A

shortened example of the PhantomCSS test script is presented on Listing 4.1. After all the screenshots are taken, PhantomCSS compares the screens against reference images and generates difference images for failed tests. Finally, the test results are printed into standard output or exported into a xUnit report.

Reference images are taken on the first execution and stored on disk for later use. PhantomCSS supports individual element testing.

```
/* Navigate to the mobile subscriptions page */
casper.then(function() {
  casper.open(config.baseUrl+'#!/matkapuhelinliittymat');
});

/* Wait 3s, then take a screenshot from Reilu box */
casper.then(function(){
  casper.wait(3000, function() { phantomcss.screenshot('#reiluBox', 'mobile
      subscriptions reilu box');  })
});

/* Take a screenshot from navigation header */
casper.then(function(){
  phantomcss.screenshot('header', 'mobile subscriptions header');
});

/* More tests ... */
```

Listing 4.1: Example of mobile subscription page tests for PhantomCSS

4.2.2 Huxley

Huxley [8] is another tool for testing web applications for visual regression. It is an open source project written in Python and powered by Selenium WebDriver. The tool is based on recording a set of tests and capturing reference screenshots that are used to verify the UI.

The Huxley test suite is defined in a *Huxleyfile* (Listing 4.2), which lists all the pages to be tested. The actual tests are then recorded by running Huxley in record mode. In record mode, Huxley opens a Selenium WebDriver enabled browser session and records all the clicks and commands given by the developer. Huxley writes these recordings into files, each test under its own directory. Finally, Huxley replays all the tests and stores the screen captures as reference images.

After the reference screen captures are taken, Huxley can be executed in a playback mode to replay the recorded tests and to verify the UI screen captures. Huxley verifies the whole page and does not support any method of testing individual elements. If a test fails, Huxley immediately prints out

an error to the standard output and stops the execution. For a failing test, Huxley creates a difference image displaying the perceived defect.

```
[subscriptions]
url=http://10.0.2.2:9601/#!/matkapuhelinliittymat
screensize=1024x1768
sleepfactor=0.5
```

Listing 4.2: Example of a Huxleyfile that defines the test for mobile subscriptions page

4.2.3 Needle

Needle [14] is an open source visual regression test tool. Like Huxley, Needle also employs Selenium WebDriver to navigate and interact with the web page and capture screenshots of the current state. Needle executes tests over Nose [15], a unit test framework for Python.

```
class MobileSubscriptionTest(NeedleTestCase):

  def setUp(self):
    self.driver.get(baseUrl+'#!/matkapuhelinliittymat')
    self.driver.find_element_by_id('reiluBox')

  def test_navigation(self):
    self.assertScreenshot('header', 'subscriptions_navigation')

  def test_reilu_box(self):
    self.assertScreenshot('#reiluBox', 'subscriptions_reilu_box')

  def test_item_without_campaign(self):
    extraInfoButton = self.driver.find_element_by_css_selector('#
        mobileSubscriptions .shopItem:nth-child(2) button.toggleExtraInfo')
    extraInfoButton.click()
    self.driver.find_element_by_css_selector('#mobileSubscriptions .shopItem
        .expanded')
    self.assertScreenshot('#mobileSubscriptions .shopItem.expanded',
      'subscriptions_item_without_campaign')
  ...
```

Listing 4.3: Example Needle test case for mobile subcriptions page.

Needle works like PhantomCSS: the developer writes Needle test cases and executes the tests once to capture reference images. The tests can then be re-run to verify the application UI for visual regression. The tests are written in Python, extending Nose unit test framework. The developer can interact with the page by using Selenium WebDriver. Needle then offers methods for verifying screen captures of either individual elements or the

entire page (see Listing 4.3). As Needle runs over Nose, it supports different test reports. However, Needle does not generate difference images for failing tests.

4.2.4 Depicted

Depicted [4] differs from the other tools for not being a library or test tool. Instead, Depicted is a runnable server that offers RESTful interface to run visual regression tests.

Depicted separates test executions to builds and releases, where a build represents a project, web site or application under test. Each build consists of several test suite executions, Depicted releases, that in turn consist of several tests.

Depicted offers a web UI to browse the releases and define the reference versions of each test (Figure 4.4). Developer can view the test results and mark failing tests as "good" or "bad", depending whether the change was desired or not.

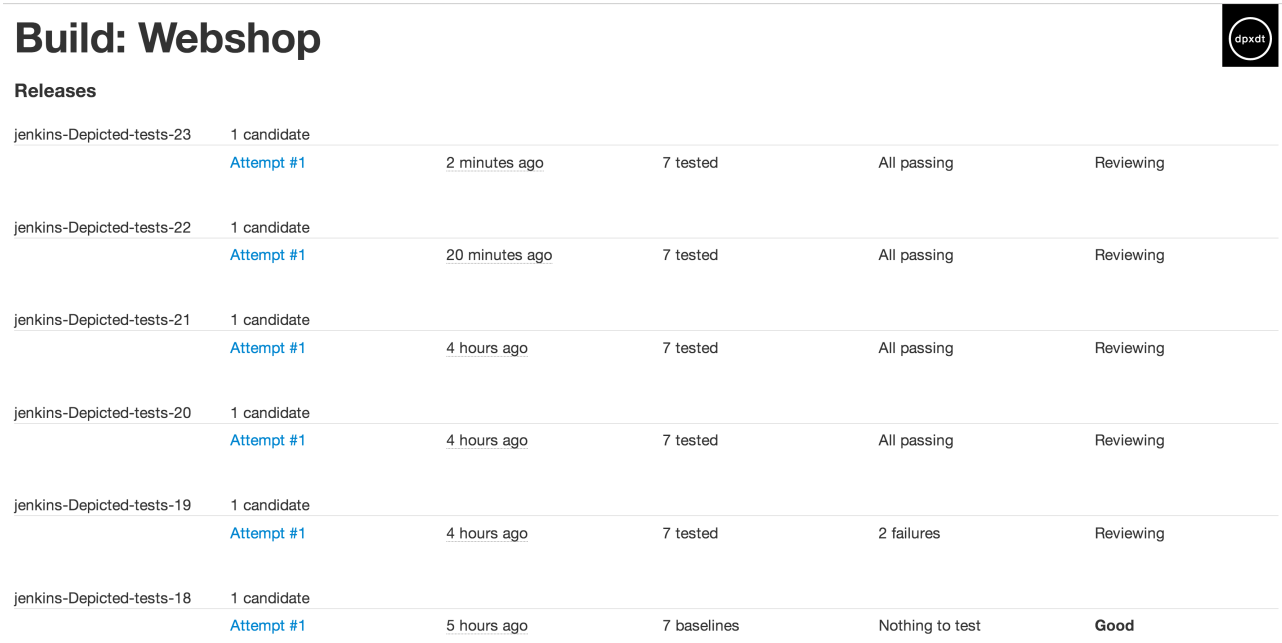


Figure 4.4: Depicted user interface showing the recent releases for the case application.

Depicted allows the developer or the CI server to define URLs to be

tested and to request a test execution using the RESTful interface. Depicted server then captures a screenshot of each page by using PhantomJS headless browser, and compares the result to the reference image. Depicted supports only full page comparisons without any interaction with the tested page.

4.3 Evaluation

To evaluate the existing visual regression tools, we used each tool to test our case application. We injected a set of faults to the case application and tested the tools one at a time. Next, we will review the findings on each tool and analyze the advantages and disadvantages of each tool. Finally, we will present a summary of the features and review the test results.

4.3.1 PhantomCSS

We found PhantomCSS to be a straightforward tool for visual regression testing, however it is cumbersome to use due to a few problems with the PhantomJS headless browser.

PhantomCSS is installed by fetching the code from the GitHub repository and implementing the tests directly over the examples. Since the tool cannot be imported into a project as a library, it becomes coupled with the test suite and is generally more complicated to use and update in the future.

We wrote a test suite that consists of eight JavaScript files: one *runner.js* to run the test suite and manage PhantomCSS, and seven separate files that contain tests for each page. With a separate *runner.js* wrapping the test startup and PhantomCSS test comparisons, the actual test scripts became readable containing mainly CasperJS navigation logic. However, as the test scripts are mostly asynchronous, they should be kept rather simple to avoid nested callback hierarchies caused by complex navigation on the target application.

Most significant disadvantages with PhantomCSS were related to the use of PhantomJS headless browser. We quickly found out that, although the browser runs on multiple operating systems, it also renders the content differently for each operating system. Thus the screen captures taken on the CI server running Linux were not even close to being pixel perfect in comparison with similar screen captures in the Mac OS X environment (Figure 4.5). This obviously prevents the developers from running regression tests compatible with each other and the CI server. Furthermore, the PhantomJS browser does not support to remotely executing the tests.

Figure 4.5: Example of an image captured by PhantomCSS with PhantomJS browser on two different operating system environments: Ubuntu Linux (*left*) and Mac OS X (*right*). The difference is clearly visible on the font rendering and it affects the element positioning and dimensions.

Another PhantomJS related problem occurred with the use of HTML5 local storage in the case application. PhantomJS runs each browser session under the same configuration and shares assets such as local storage and session data. This becomes problematic when the tested application employs the local storage data and for example displays different elements on the page. To solve this, we had to manually clear the local storage data every time before the tests navigate to a new location. Moreover, this solution works only partially and would lead to a race condition where two or more PhantomJS browser sessions would compete over the same local storage data if run in parallel.

Following the interoperability limitations of the PhantomJS browser, we cannot store reference images together with the test suite code into the version control repository because the images would be incompatible with developers' environments and the CI server. Instead we have to store reference images locally on each machine, which makes it difficult to develop and debug the tests.

As PhantomCSS captures the reference images only if no previous reference images exist, we have to clear the reference image directory and run the PhantomCSS test suite once to capture new baseline images. To achieve this on the CI server, we implemented a separate Jenkins job to remove the old images and execute the initial test run to capture the baseline images.

For failing tests, we configured Jenkins to store the difference images for each test run so that the developer can review the images and identify the faults. Whenever the software UI is changed on purpose, the developer has to review the saved difference images and run the Jenkins job to reset the reference screenshots. Since it is impossible to reset the baseline images for only the failing tests, it the developer has to review all the reference images manually and verify that they are correct after the reset.

4.3.2 Huxley

Huxley can be installed as a standard command line tool by using a Python package manager. At first we tried to use Huxley to test the entire case application as intended, however this appeared to be problematic. We were facing several difficulties while trying to record the tests and capture reference images, and thus ended up writing a simple *Huxleyfile* defining the tests for only the content pages, leaving out the checkout-views.

In our experiment, we found many disadvantages when performing the regression testing with Huxley. Firstly, Huxley's record-mode supported interaction only partly. We were able to interact only with the initially visible part of the screen. For example, we scrolled down on the mobile subscriptions page to click one of the additional info boxes open. Huxley recorded the click but misplaced the coordinates and failed to play it back correctly when capturing the reference images. We managed to overcome this problem by defining the initial screen size to be larger, but the problem complicates the use of the tool considerably. The issue becomes more significant if the developer has to re-record the tests often.

Secondly, the use of local storage in the case application caused problems. When running Huxley, it starts one browser session to replay all the tests and the local storage is not cleared between the tests. If an individual test on a larger test suite adds a product to the shopping cart, the next test will have the same product in the cart. This will cause false positive failures for example if the test execution order changes randomly. For this reason, we decided to exclude the testing of the shopping cart and checkout process. The problem is similar with the findings from PhantomCSS (Section 4.3.1), although the issue is caused by Huxley's method of invoking the test suite and cannot be overcome by writing additional scripts to clear the local storage.

Additionally, we observed that different browsers and operating system configurations caused differences in the font rendering and element positioning, similar to the PhantomJS problems encountered previously (Figure 4.5). To allow pixel perfect screen captures between different environments, we installed Selenium server on the CI server, executed all the tests remotely and stored the images to the version control repository. However, as the developer cannot record Huxley tests remotely, we created separate configurations to allow the developer to run the tests also on a local Selenium server.

Huxley generates difference images (Figure 4.6) for failing tests and we configured Jenkins to store the images on each build. Huxley does not support individual element testing, thus each screenshot covers the entire web page and will fail regardless of the failing component or how significant the defect is. This makes it harder to spot the actual errors, since visual defect

on a common widget will cause test failures on multiple pages. This becomes emphasized in spatial errors, such as the injected fault number 2. (Table 4.1) where entire page content is shifted downwards.

Figure 4.6: Example section of a difference image produced by Huxley.

Finally, the most significant issue with Huxley was the false positives it detected on our tests. We found that Huxley was completely dependant on the reference images as the CI would always produce the same false positives for each reference image set. In the example, Figure 4.7, we can see how Huxley falsely detected an error in a linear background gradient. Similar false positives were observed throughout the experiment, which seriously hinder the usability of the tool.

When the UI of the tested software changes, the developer has to remove all the existing reference images and re-run Huxley in record mode. As previously noted the recording appeared to be problematic and prone to accidental errors. Therefore, similarly to PhantomCSS, the developer has to review all the reference images after they are re-recorded.

Figure 4.7: Example of a false positive detected by Huxley.

4.3.3 Needle

Needle was installed easily with Python package manager and the tests were easy to setup. We wrote separate test classes for each page to be tested. The test scripts consisted mainly of asynchronous Selenium navigation logic that navigates through the case application and tries to verify the UI state.

Due to the dissimilarities in rendering with different operating systems and browsers, we decided to use similar setup with Huxley and execute Selenium WebDriver remotely on the CI server. We stored the reference images to the version control repository. To reset the images, developer has to first remove the existing images, run the initial capture and commit the changed images to the repository.

Similarly to Huxley, Needle suffered from occasional false positive test results. However, we found the missing difference images to be the most significant disadvantage. Needle does not generate difference images for failing tests and therefore the developer has to somehow manually verify the view, which makes it difficult to find the actual problem that caused the test to fail.

4.3.4 Depicted

Even thought Depicted is written in Python, it does not support standard Python packages and must be installed manually. We installed Depicted server on the CI server by cloning the version control repository and initializing the Git submodules of the project.

Depicted has a few disadvantages. First, the client side installation was found to be somewhat intractable. Depicted offers example Python scripts for crawling an entire site or comparing two URLs. Since the REST interface of the server is quite complicated, the scripts utilize a set of client-side Python classes that help to set up the requested test suite. However, the client side library is coupled in the same Git repository with the Depicted server and requires Git submodules. Because of this, we could not create a separate repository with all our test scripts but had to modify the example scripts for our purpose and store them on the CI server.

Second, the Jenkins integration suffered from the lack of actual build results. Jenkins integration was done by simply executing the modified test scripts with a Jenkins build. Since the Depicted REST interface does not offer any method for fetching details of a release, the scripts would create a new Depicted release, assign a test suite for it, and exit without any knowledge of the build status. Consequently we cannot display the true result of a build in Jenkins but the developer must always view the build status from the

Depicted web UI. This complicates the developing process because all the data should be quickly available from the CI server.

Finally, we encountered a major bug in the Depicted server software. After a CI server restart, the SQL database was in an erroneous state and the Depicted server would not start. To fix the issue, we had to completely clean the database and start from the beginning.

Even though the integration with the Jenkins server was incomplete, we found the overall workflow of the Depicted tool to be a great advantage in comparison with the other tools. To repair an outdated test, the developer has to simply verify the changes from the web UI. Accepting changes for individual failing tests was easy to use and the web UI provided good visualizations on the detected difference. Verifying releases through the UI is definitely a more usable solution than resetting the reference images on the CI server or version control.

4.3.5 Findings

Summary of the features supported is presented in Table 4.2. PhantomCSS and Needle are the only to support individual element testing and proper interaction and navigation within a page. Huxley had insufficient support for interaction and Depicted did not have support at all.

Needle and Huxley utilize Selenium WebDriver and thus support multiple browsers. PhantomCSS and Depicted both utilize PhantomJS headless browser. Jenkins integration was properly supported by Needle and PhantomCSS as both were able to store the test results as xUnit report.

From all of the evaluated tools, only Depicted supported individual reference image reset. With Depicted, the reset can be easily performed through a web UI that visualizes the defects. PhantomCSS, Huxley, and Depicted provide difference images to visualize the defects on failing tests. Needle, on the other hand, does not, which complicates the developer's workflow considerably compared with the other tools.

Fault detection results are visible in Table 4.3. PhantomCSS and Needle performed with the best results, Needle finding all of the faults and PhantomCSS totaling nine out of ten. Huxley supported interaction only partly and found seven of the faults, while Depicted found only the faults that do not require interaction.

Test suite execution times vary from an average of 19 seconds for Huxley to an average of 138 seconds for Needle (Table 4.4). The difference is substantial but not excessive. For visual regression tests, the execution time of little over 2 minutes for Needle is sufficient for our case application and

considering that the other integration tests for the same application consume more than triple the time.

	PhantomCSS	Needle	Huxley	Depicted
Full page testing	√	√	√	√
Individual element testing	√	√		
Page interaction	√	√		
Test reporting	√	√		
Difference images	√		√	√
Individual reference image reset				√
Multiple browsers		√	√	
Headless browsers	√			√

Table 4.2: Feature comparison

Fault	PhantomCSS	Needle	Huxley	Depicted
1	√	√	√	√
2	√	√	√	√
3	√	√	*X*	*X*
4	√	√	*X*	*X*
5	√	√	√	√
6	√	√	√	√
7	*X*	√	√	√
8	√	√	√	√
9	√	√	*X*	*X*
10	√	√	*X*	*X*

Table 4.3: Fault detection results for each injected fault.

PhantomCSS	Huxley	Needle	Depicted
64	19	138	130

Table 4.4: Average test runtime in seconds.

All of the tested tools take screenshots from the UI and compare them against reference images that have been captured during previous test runs. We identified three common problems for this type of pixel-perfect comparison approach. First, the maintenance of reference images is difficult. Reference images for PhantomCSS, Huxley and Needle were stored into either the version control repository or on the CI server. When new tests are

added or existing tests are updated, the developer needs to delete all existing images and re-run the whole test suite to capture new baselines. Second, the interoperability between different environments and browsers is a problem when comparing screenshots. Images taken on different operating systems are incompatible as the text and fonts are rendered differently. Finally, the false positives will distract the workflow and cause the developers to lose their trust in the test system.

In addition to the common problems of comparing the screenshots, we encountered several problems with the tools themselves. Depicted suffers from usability problems with the client library and a severe bug leading to data loss. Huxley and Depicted are unable to test the application thoroughly and therefore are not adequately applicable to our visual regression testing problem. However, we found the Depicted UI to be an easy and advantageous method to manage the test results and reference images.

In conclusion, only Needle and PhantomCSS performed sufficiently and managed to test the entire case application. The ability to interact with the page is the single and the most determining factor on how the tools succeed to detect the faults. The missing shadow in the fault number 7 is too indistinguishable a change and was not detected by the image comparison algorithm of PhantomCSS. This could be fixed by slightly adjusting the difference detection.

It is certain that the management of reference images has a huge impact on the problem of repairing outdated tests and coping with the changes in the tested software. Since the capture and replay tools are unable to prevent the tests from becoming outdated, the testing tools must provide quick and easy manner to verify the changes. PhantomCSS, Needle, and Huxley required plenty of manual work in comparison to Depicted, which provided an easy access to the reference images through its web UI.

Chapter 5

New tool for visual regression testing

Using the knowledge gained from the background studies and evaluation of the existing tools, we created a new tool, named *Giffidiffi*, to perform the visual regression testing. In this chapter, we will describe the design, architecture and implementation of the tool as well as evaluate and analyze the advantages of the tool in comparison with the previously evaluated tools.

5.1 Design

The primary goal for the tool is to allow developers to perform visual regression testing of a web application. As we outlined in Section 1.1, the tool should be able to perform the regression testing in Continuous Integration environment. Additionally, we decided that the tool should be released as an open source project.

The existing testing tools rely on a capture and compare approach to test the visual appearance changes. The tools take screenshots of the UI and verify them against reference images that have been captured during previous test runs. We will base our design on a similar method and perform screenshot comparison to test the layout for visual regression issues.

Since we will utilize the same screenshot comparison technique as the existing tools, we can benefit from the evaluation findings in Section 4.3. The evaluation exposed three common problems for comparing screenshots: the maintenance of reference images, the interoperability between different environments and browsers, and the false positive test results.

The first issue, the maintenance of reference images, is related to a common issue of user interface tests becoming easily outdated or broken when

the tested software is modified [38]. We will especially focus on this and design the tool so that it provides an easy and fluent manner to update and remove the outdated tests.

To overcome the second issue, we will design the tool to decouple the test execution and the reference image management. This will improve the interoperability between different environments and browsers since the tests can use reference images depending on the test environment. For example developers and the CI server can use different baselines and different browsers if required.

Finally, possible false positive test results are avoided by allowing the developer to adjust the image difference detection algorithm sensitivity. Furthermore, the tool should offer helpful visualizations for the developers to review the failing tests and defects found.

Based on these requirements, we define the following high-level design goals for the new tool:

- Test web application layout for visual regression,
- Provide test reporting for CI servers,
- Allow easy update and removal of broken and outdated tests,
- Decouple test execution and reference image control,
- Allow to adjust the image difference detection sensitivity,
- Provide visualization of the test failures,
- Release as an open source project.

Additionally, we will focus on integrating the tool to the developer's workflow and making it easy to use. The workflow follows the regression testing process (see Section 3.1) closely:

1. Developer commits changes to the version control repository,
2. CI server builds the application and executes the unit tests,
3. CI server executes visual regression tests as part of the integration test suite,
4. Developer checks the build results from the CI server,
5. Developer will review the screenshots of failing tests, if any, against corresponding reference images,

6. Developer will mark failing tests that are desired result of the code change, if any, as accepted changes,

7. Developer will fix all remaining failing tests, if any, and start from the step 1.

5.2 Architecture

Giffidiffi-tool consists of three main components: *test runner*, *server*, and *user interface*. The full architecture is illustrated in Figure 5.1.

Test runner is a software that executes Giffidiffi test suites. The test runner interacts with a web application, captures the UI state into a screenshot and sends it to the Giffidiffi server. The server is a typical web server that runs the actual test evaluations and stores all test related data, such as captured screenshots, reference images, and test results. Once the server receives screenshots of the web application under test, it compares the image with previously captured baseline image, stores the result into a database, and sends it back to the the test runner. When all of the tests in a test suite are executed, the test runner generates a test report that states the results.

The third component, user interface, is used to visualize the test results. The UI lists all the test results and visualizes the failed tests and defects found. Additionally, the UI allows the developer to accept the changes in the tests to reset the baseline images.

The server stores the test results and captured screenshots into a database. The image processing and comparison of two images is performed with an external GraphicsMagick[1] software.

[1] `http://www.graphicsmagick.org/`

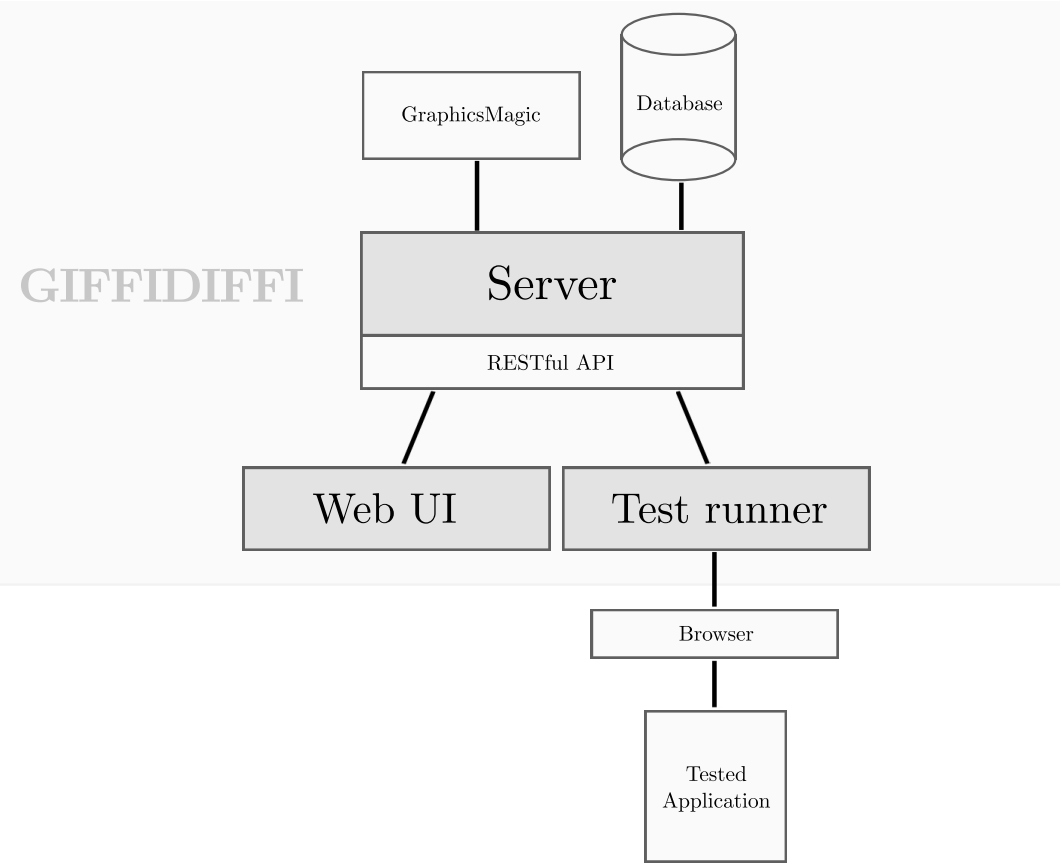


Figure 5.1: Giffidiffi-tool architecture

The reference image flow is illustrated in Figure 5.2. The initial build is always successful and the captured images A_0 and B_0 are used as reference for future builds. In *Build 2*, the test runner captures image A_1 and the test fails because it does not match with the reference image. The tested application is modified again in *Build 3* and the developer verifies the change in the test A using the Giffidiffi UI. *Build 4* introduces a new test Z and the captured screenshot Z_0 is automatically used as a reference image. Similarly the test A is removed from the test suite in *Build 5* and the build is successful. If the test would be re-introduced in the future, the image A_2 would be used as a reference.

The client-server architecture enables to overcome the first two common visual regression testing problems. First, the reference images are stored on the server together with the test results. The server can visualize test results with a web UI and allow developers to maintain the reference images

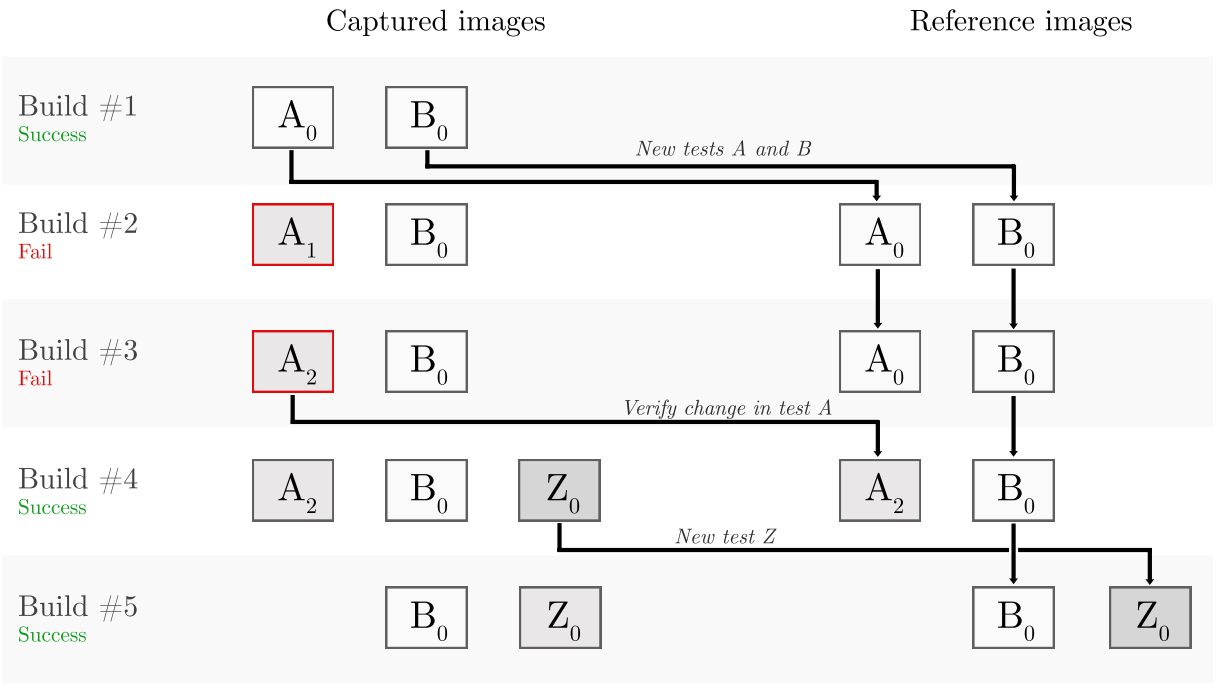


Figure 5.2: Reference image flow

directly from the UI. When a test fails, developer can check the cause from the UI and verify whether the layout change is desired or not. If the change is accepted, the next execution of the same test will use the new screenshot as a reference image in the comparison. Naturally, the developer can also revoke the accepted change and use the previously accepted reference image. Additionally, adding and removing tests does not require any action of the developer regarding the reference images. When a new test is introduced into a test suite, the first execution for the test will always be successful and set the reference image. To remove a test from a test suite, developer can just leave it out, since Giffidiffi-tool will not try to maintain a list of tests that should be included in builds.

Second, the server does not restrict the choice of the browser or capturing technology since it receives the test data from the client. The test execution is decoupled from the reference images, therefore improving the interoperability. The Giffidiffi server can have separate builds for different environments, for example the CI server running on Linux machine can use different reference images than the developers running OS X. Test runner can capture the screenshots with any technology or browser preferred, hence

allowing developers to customize and configure the runner if needed.

The third common problem, false positive test results, can be avoided by having the image comparison tolerance as a configurable option on the test runner. The tolerance can then be adjusted if false positives are detected. However, if the tolerance level is set too high, Giffidiffi may be unable to detect the actual errors.

The tool integrates into the Continuous Integration environment easily by providing adequate test reporting. The test results are immediately available since the server returns the result for each test in the HTTP answer. The test runner then aggregates the results and creates a test report for the CI server.

5.3 Implementation

Our Giffidiffi implementation consists of separate components for the server, web UI, and test runner. Next, we will discuss the implementation details and introduce the technology decisions we made.

5.3.1 Server

Giffidiffi server is implemented on Node.js[2] application platform using CoffeeScript as the programming language. We utilize Express[3] framework, which provides methods for creating asynchronous Node.js web applications, as well as Bacon.js[4] and Lodash[5] libraries.

We selected SQLite[6] database to preserve the test results and captured screenshots. Since SQLite stores the data into a single database file and does not require a separate daemon software to run, Giffidiffi server can be easily installed without the need for a pre-installed database software. Moreover, SQLite also provides a simple inmemory database that is extremely useful for running tests.

The database schema is simple and consists of only two tables: `documents` and `attachments`. The `documents`-table holds all the test data on key, type, and value columns. Key and type are simple strings and value contains the stored document as JSON structure. All screenshots are stored into the

[2] `http://nodejs.org/`
[3] `http://expressjs.com/`
[4] `http://github.com/baconjs/bacon.js/`
[5] `http://lodash.com/`
[6] `https://sqlite.org/`

`attachments`-table within similar key, type, and value columns, where the value column contains the image as binary data.

The test results are modelled with three different type of documents: *projects*, *builds*, and *test*. The data model is illustrated in Figure 5.3. One Giffidiffi server can host several projects. Each project is identified by a unique name and it contains a number of builds. Builds are identified by a number and are always related to a specific project. Each build has a set of individual tests that define the status of the build. The tests are identified by a name, which is unique within each build, and contain the result of the test as well as pointers to the screenshots and generated difference images. Build is considered as failed when at least one of its tests have a failing status and similarly successful when when all its tests are successful or verified.

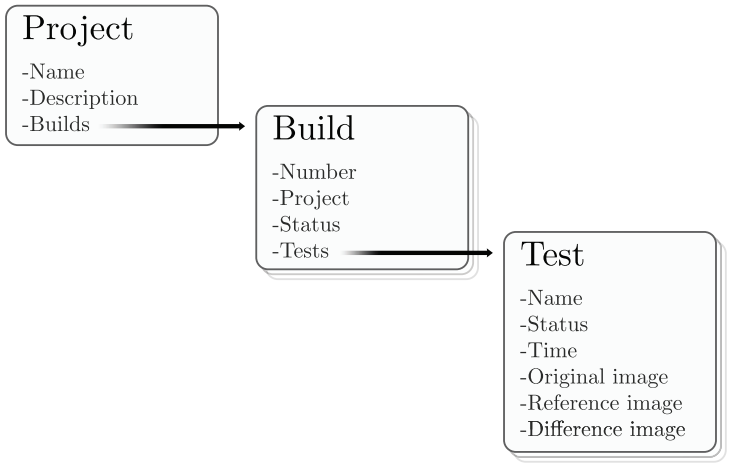


Figure 5.3: The Giffidiffi data model.

The Giffidiffi server provides an API to control and fetch the data. The interface resurces and their descriptions are listed in Table 5.1. Test runners use the API to create builds and add tests to the builds. When a new test is added into a build, the server routes the HTTP request to a `Controllers.test.runNewTest` function and begins by storing the captured screenshot to the SQLite database. The image data is included as multipart data in the HTTP POST request. Next, the server tries to find a reference image for the new test by searching a successful test execution from the previous test results. If a reference image is found, it is used to calculate the difference and to decide whether the test is successful or not. Finally, the server stores the test result to the database and returns a simple success/fail result. The process is illustrated as a flowchart in Figure 5.4.

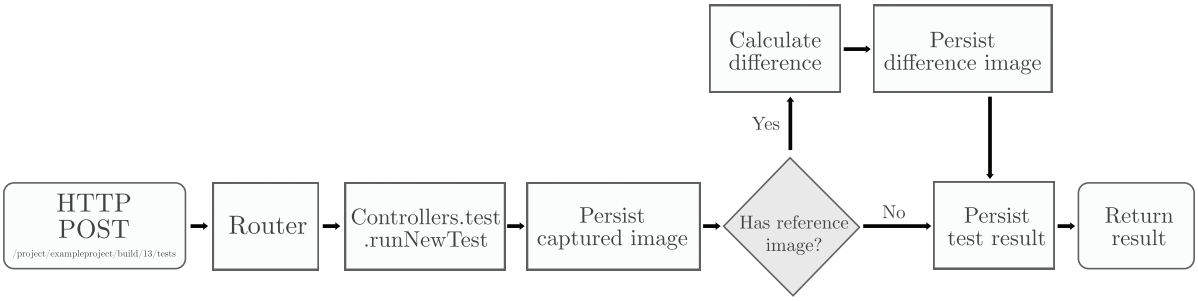


Figure 5.4: Process of adding new test to a build.

The image comparison is done with GraphicsMagick[7] software, which features a ready-made tool to perform the comparison and output a difference image. GraphicsMagick is fast in comparing images and easy to use for our purpose.

[7] `http://www.graphicsmagick.org/`

/api/project	
GET	Get all projects.
POST	Create a new project.

/api/project/{projectName}	
GET	Get project with `projectName`.
PUT	Update project `projectName`.
DELETE	Delete project `projectName`.

/api/project/{projectName}/build	
GET	Get all builds for project `projectName`.
POST	Create a new build for project `projectName`.

/api/project/{projectName}/build/{number}	
GET	Get the build data for build `number`.

/api/project/{projectName}/build/{number}/tests	
GET	Get all tests for the build.
POST	Add a new test to the build.

/api/project/{projectName}/build/{number}/done	
POST	Mark build as complete, thus do not accept new tests.

/api/project/{projectName}/build/{number}/tests/{testName}/good	
POST	Mark individual test as verified, thus set the test status to `success`

/api/project/{projectName}/build/{number}/tests/{testName}/bad	
POST	Mark individual test as failed, thus set the test status to `fail`

/api/project/{projectName}/build/{number}/tests/{testName}/{image}	
GET	Fetch the requesed `image` for the test. Available image types: `original/reference/difference`

Table 5.1: List of all the Giffidiffi server API resources.

5.3.2 Test runner

Since Giffidiffi server itself does not capture any screeshots or execute the test suites, we created a separate test runner. First, the test runner creates a new build to the project by posting a request to `/api/project/{projectName}/build`.

After the build is created, the test runner executes the test suite and sends each screenshot to the `/api/project/{projectName}/build/{number}/tests` resource with correct a project name and build number in place. When all the tests are executed, the build needs to be marked as completed to prevent more tests from being added to it. The test runner does this by calling the `/api/project/{projectName}/build/{number}/done` interface.

We implemented one test runner to conduct the actual testing. The test runner is an extended version of the visual regression testing tool Needle [14], which was evaluated in our evaluation (see Chapter 4). Needle provides a solid base to work on and it is easy to extend by forking the Git repository. Needle is written in Python 2.7 and it extends a popular unit test framework Nose[8], which provides support for creating test suites and executing actions before and after tests. `GiffidiffiTestCase`, our extension of the standard unit test `TestCase` class, provides Selenium WebDriver support for the test and allows the test to send screenshots to the Giffidiffi server for evaluation.

Since we extended Needle, the test scripts have much in common. Listing 5.1 shows a partial test script for our case application. The test script is almost identical to the Needle equivalent presented in Listing 4.3. However, Giffidiffi syntax for executing the assertions is different.

```
class MobileSubscriptionTest(GiffidiffiTestCase):

  def setUp(self):
    self.driver.get(baseUrl+'#!/matkapuhelinliittymat')
    self.driver.find_element_by_id('reiluBox')

  def test_navigation(self):
    self.assertVisualDifference('subscriptions_navigation', 'header')

  def test_reilu_box(self):
    self.assertVisualDifference('subscriptions_reilu_box', '#reiluBox')

  def test_item_without_campaign(self):
    extraInfoButton = self.driver.find_element_by_css_selector('#
        mobileSubscriptions .shopItem:nth-child(2) button.toggleExtraInfo')
    extraInfoButton.click()
    self.driver.find_element_by_css_selector('#mobileSubscriptions .shopItem
        .expanded')
    self.assertVisualDifference('subscriptions_item_without_campaign',
      '#mobileSubscriptions .shopItem.expanded')

  ...
```

Listing 5.1: Example Giffidiffi test case for mobile subcriptions page. Differences to Needle syntax are highlighted in light green.

[8] `http://nose.org/`

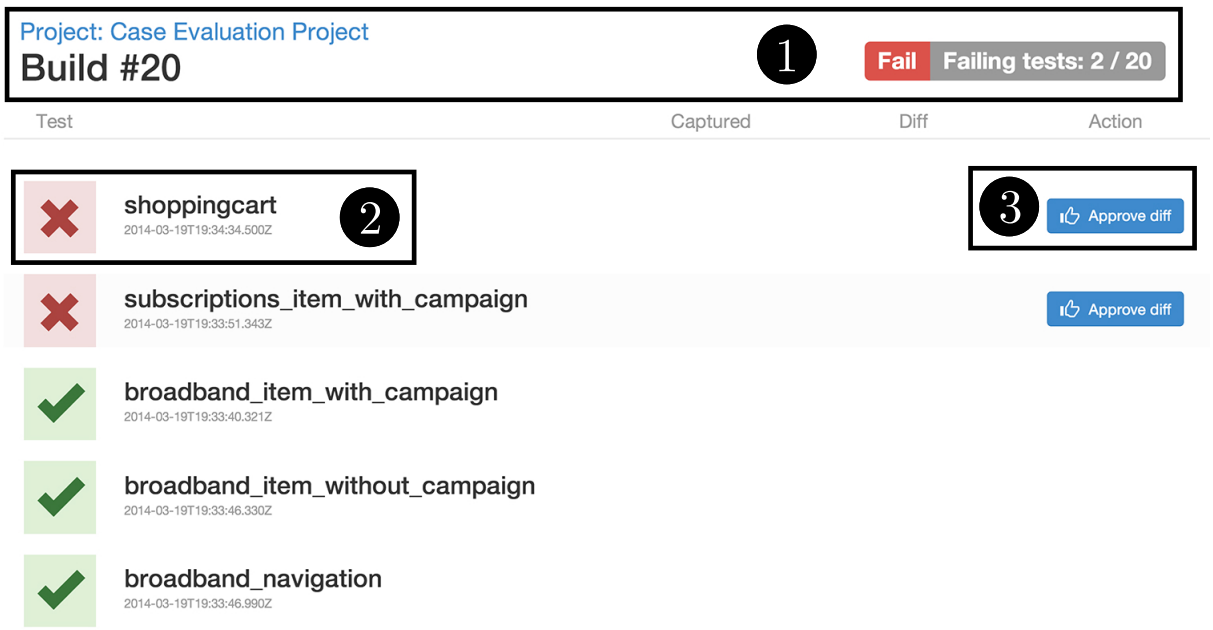


Figure 5.5: Giffidiffi UI tests view listing multiple test resuls. The rows are color coded for failing and successful tests in red and green respectively.

5.3.3 User interface

The single-page web application of Giffidiffi provides two simple features: visualize the test results, and accept the changes in tests. We decided to implement the UI in CoffeeScript and utilize Bacon.js and Lodash libraries, similarly to the server-side. CoffeeScript code is compiled into JavaScript with Grunt[9] to execute the UI logic on the browser.

The UI provides three views to visualize the tests and test results: *projects*, *builds*, and *tests*. The projects-view serves as the application root and lists all the different projects available on the server. The builds-view shows all the builds for one project, with the latest one on top of the page. Additionally, the aggregated status for each build is visible in the builds-view, therefore allowing the user to rapidly see the status of the project. Finally, the tests-view (see Figure 5.5) illustrates the individual test results for selected build. The build details and result is displayed on the top (item 1 in Figure 5.5). Each test has a row that displays the test result on the left and test name and details on the center (item 2 in Figure 5.5). For each failing test, the UI

[9] `http://gruntjs.com/`

displays a button to accept the change (item 3 in Figure 5.5). When a test row is clicked, the UI visualizes the captured screenshot, reference image, and difference image below the row (see Figure 5.6). Hence, the developer can view the captured images and verify the change from the difference image.

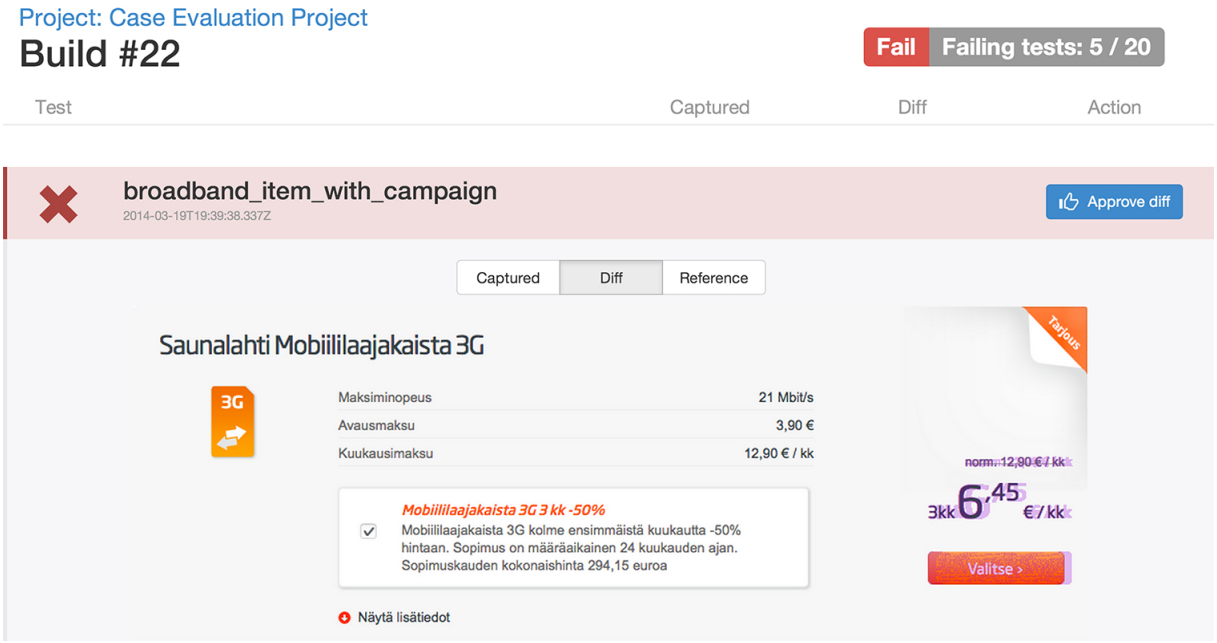


Figure 5.6: The UI can visualize individual test results. The image shows the difference between the captured and the reference image in dark violet overlay.

5.4 Evaluation

We evaluated the new Giffidiffi tool with the same methods as in the evaluation in Chapter 4. We installed Giffidiffi server on the CI machine together with the Python test runner, and created a new Jenkins job for the Giffidiffi tests.

Giffidiffi performed well in the feature comparison, having nine out of ten features, lacking only the support for headless browsers (see Table 5.2). Currently, it has a test runner implementation only for Python and Selenium WebDriver combination, thus allowing test execution on multiple browsers. However, our design leaves open the possibility to create a separate test

runner for a headless browser.

Comparing the feature set with PhantomCSS and Needle, which were found to be adequate for testing the visual regression, Giffidiffi offers the advantage of resetting individual test images from the web user interface. For PhantomCSS and Needle, the developer needs to remove the existing images and re-run the test suite to accept the changes to the UI. Additionally, Needle does not generate difference images for failing tests, which complicates the developer's work when reviewing test results and verifying changes.

	Giffidiffi	**PhantomCSS**	**Needle**	**Huxley**	**Depicted**
Full page testing	√	√	√	√	√
Individual element testing	√	√	√		
Page interaction	√	√	√		
Test reporting	√	√	√		
Difference images	√	√		√	√
Individual reference image reset	√				√
Multiple browsers	√		√	√	
Headless browsers		√			√

Table 5.2: Feature comparison including Giffidiffi tool.

The fault detection results are revisited in Table 5.3, which shows that Giffidiffi was able to found all the seeded faults. The complete coverage was possible since Giffidiffi can review also parts of the UI that require interaction to become visible. The results are similar to those of Needle and PhatomCSS. However, we had the ability to adjust the image comparison tolerance values for Giffidiffi, which made it possible to ensure that the tool finds all the differences in the images. Because of this, Giffidiffi was able to detect the missing shadow on fault number 7 (see Table 4.1), which PhantomCSS was unable to detect. An issue worth noting is that adjusting the tolerance level too low will cause false positive test results.

The average execution time for Giffidiffi test suite was 124 seconds (see Table 5.4), which is on the same level with the other tools and can be considered as sufficient.

Giffidiffi combines the advantages of Needle and Depicted. The support for interaction and individual element testing is combined with the web UI that allows the developers to manage the reference images. Giffidiffi test runner is easily installed with Python package manager. The server is installed by fetching the Git repository and running install scripts. The difference images, which were missing for Needle, are now provided through the web UI. At the same time, Giffidiffi will support CI server by reporting the actual

Fault	**Giffidiffi**	**PhantomCSS**	**Needle**	**Huxley**	**Depicted**
1	√	√	√	√	√
2	√	√	√	√	√
3	√	√	√	*X*	*X*
4	√	√	√	*X*	*X*
5	√	√	√	√	√
6	√	√	√	√	√
7	√	*X*	√	√	√
8	√	√	√	√	√
9	√	√	√	*X*	*X*
10	√	√	√	*X*	*X*

Table 5.3: Fault detection results for each injected fault including results for Giffidiffi tool.

PhantomCSS	**Huxley**	**Needle**	**Depicted**	**Giffidiffi**
64	19	138	130	124

Table 5.4: Average test runtime in seconds including results for Giffidiffi tool.

test results immediately, unlike Depicted, which required an additional check from the Depicted server.

Giffidiffi has also a few disadvantages. First, the client-server architecture complicates the test setup. Instead of a simple standalone test tool, Giffidiffi requires the configuration of a test runner, Giffidiffi server, and possible Selenium server. All of these components must operate and have proper configurations, such as remote host and port information, in place. Debugging possible failures in this setup is a lot more difficult than for example for a single component PhantomCSS tool.

Second, it is unknown how the server will perform when it is being used by multiple projects and development teams since we did not have the possibility to test the tool with several developers or in a multi-project environment. The application code is still in a prototype phase and may therefore contain some design flaws that were still unknown to us during this study.

Chapter 6

Discussion

This chapter discusses the results and implications of this study as well as the possible future work. First, we will discuss the implications that the evaluation results have. In addition to this, we will also consider the effect that the new Giffidiffi tool may have from the developers' perspective. Second, we will analyze the research methods used and identify possible limitations on the findings. Finally, we will discuss the future work and research regarding to visual regression testing.

6.1 Implications of the results

We evaluated four existing visual regression testing tools and analyzed their advantages and disadvantages. We were able to overcome most of the found bugs and problems in the tools. However, problems related to the general workflow are more difficult to fix with an additional script or other small modification.

We found that keeping tests up to date with the tested application is the most significant issue with the existing visual regression testing tools. When a software is under development, it is modified constantly, especially when using agile development practices. In this type of environment, regression tests should require as little manual work as possible. If the tests are difficult to keep up to date, they will more probably deteriorate and eventually even become more harmful than helpful. Difficult reference image management causes more work for the developers to keep the tests up to date, which will lead to the degeneration of the tests. Thus, we found that the process of adding, removing and updating reference images should support the developers' workflow and be as simple as possible.

However, difficulties in reference image management do not necessarily

reflect on the ability to detect the visual regression issues. Even though Needle and PhantomCSS provided insufficient and mainly manual mechanisms to keep the reference images up to date, they were still able to detect the visual regression problems and test the entire case application. The third tool, Huxley, was considered inadequate because it lacked proper interaction support and suffered reliability problems.

Depicted was the only tool to offer a solution to update reference images. However, Depicted is capable of capturing one screenshot per page and does not provide any method of interaction with the tested application. For this reason, it is suitable for testing a set of standard web pages but not an entire web application.

Additionally, during the evaluation we found out that testing an entire page at once is an ineffective and unreliable approach. If the test compares the whole page in one screenshot, spatial errors can move the page content down and result in obscure error visualizations. Thus, it becomes difficult for the developer to detect the actual regression error. The actual fault may also be relatively small compared to the whole page screenshot and thus difficult to find from the visualization. Similarly, when a common element changes, such as the page heading, it will cause unnecessary test failures on several pages instead of failing just one test. Instead, if the tests focus on individual widgets, these problems are avoided. Naturally, creating test cases for individual widgets causes overhead which should be taken into consideration when the test design decisions are made.

The new Giffidiffi tool introduced in Chapter 5 offers several advantages in comparison with the other evaluated tools. However, the most significant advantage for Giffidiffi is its ability to support the developers' workflow more comprehensively than the other tools. Using Giffidiffi developers can test web applications for visual regression problems easily and without the need to constantly waste resources on keeping the tests up to date.

Giffidiffi tests can be fully integrated into the CI server and test results are visible immediately after each build. Maintaining the tests and reference images is easy because Giffidiffi provides a user interface to visualize test results and to accept changes on the reference images. In order to provide a full solution to test the visual regression, we also implemented a Giffidiffi test runner, which interacts with the tested application and can test complex applications.

Tests that are created with the evaluated tools are more likely to fall behind and degenerate because they are too difficult to keep up to date. With the proper CI server support and easy to use UI, the developers are more likely to update the tests and maintain the Giffidiffi build status successful.

In addition to this, the Giffidiffi architecture makes it possible to extend

and improve the tool further. The tool will be released as an open source project in GitHub[1]. Therefore anyone can use the tool free of charge and, for example, create a new test runner utilizing headless PhantomJS browser, which is not yet supported by the tool.

6.2 Review of research methods

The evaluation in Chapters 4 and 5 is based on experimenting with the tools by performing visual regression testing in a CI environment. The focus was on creating a setup equivalent to a real developing environment and collecting observations on the tools. In general, this evaluation method revealed the tools' issues and inconsistensies in the workflow very well. However, the results will hold true only for the tools which capture and compare screenshots taken from the tested application.

We gathered ten different visual errors from the development history of the case application for evaluating the tools. Because the faults were real world examples, they represented the actual use cases of the tool and we were able to trust that the results correlate with real world use. Additionally, the faults were categorized, which provided us valuable information on why the tools did not detect the faults. The results show how Huxley and Depicted failed to find those faults that required interaction with the application. However, categorizing the faults into spatial change and appearance change did not produce any additional information.

The evaluation methods have three limitations. First, the experiment was conducted with only one developer. Conducting the experiment with an entire software development team could reveal some additional issues related to shared use and interoperability. Second, the evaluation methods assume that the tested software is being developed using agile development methods and utilizing continuous regression testing with the CI approach. The test objectives and the use of testing tools is subjective for each development organization and may thus produce different results, for example, if the visual regression testing is conducted only after each software release. Third, the tested software was assumed to be a complex JavaScript application, which requires interaction support to test the entire application. A simpler web application could be tested without the support for interaction, which again yields different results.

Additionally, it should be kept in mind that visual regression testing tools are evolving constantly and this study represents only the current state

[1] `https://github.com/`

of the four evaluated tools. As discussed in Section 4.2, a few tools were excluded from the evaluation due to lack of applicability. Furthermore, it is possible that we might have missed some testing tool completely when selecting the tools to be evaluated. Different selection of tools might have produced different results.

Because the new Giffidiffi tool was evaluated with the same methods as the other tools, we could compare the tools with each other. Results show that Giffidiffi has advantages in comparison with the other tools and it provides better integration into the developer's workflow. However, there is one caveat to be taken into account. Since Giffidiffi has not been used by an actual development team, the results do not show how the advantages are perceived by the team and whether they would consider it helpful or not in a real world use case.

Moreover, the Giffidiffi tool itself is not complete, thus some improvements to the test runner and visualization UI should be made before the tool can be released as open source.

6.3 Future work

Giffidiffi is currently in a prototyping phase and requires more work to be ready for use. Before releasing the tool, the code should be reviewed thoroughly. To ensure software quality, proper end-to-end tests should be created for the test runner. Improvements on the UI should be made to display metadata regarding the tests, such as the software version of the tested application and the original build number of the reference image. This would help the developers to interpret the test results. Additionally, it should be made possible to adjust the image comparison tolerance levels for each test. This way the developers could adjust individual tests if the tests become unstable and prone to false positives.

Releasing Giffidiffi as an open source project opens new possibilities to improve the tool. If the project can reach the open source community, it is possible that they will add new features or create new test runners. Ideally, Giffidiffi would have multiple test runners utilizing multiple platforms and technologies. This would allow the developers to choose the implementation that is the most suitable for their use and environment.

Considering the visual regression testing field in general, in an ideal situation, visual regression tests could distinguish accidental and desired changes without any, or only occasional, human input. Consequently, the tests could ignore desired changes and raise a warning only if the layout is accidentally broken. Naturally, this type of artificial intelligence cannot be achieved with

pixel-perfect image comparison. Thus, future research should be done to develop more advanced visual regression detection methods, for example methods utilizing computer vision or machine learning. Computer vision could be utilized to analyze the screenshots and to identify different elements from the image, such as text, menus, and background images. Using this information in conjunction with machine learning methods, Giffidiffi could learn to classify desired and accidental changes. This would reduce the manual work required to manage the reference images and to verify changes.

Additionally, the interoperability issues could be avoided with more advanced detection methods. The pixel-perfect image comparison cannot distinguish rendering differences from actual regression issues. Since the differences caused by different operating systems, graphic cards, and browsers are mostly related to the font rendering and aliasing, they could be overcome with more advanced regression detecting methods.

Chapter 7

Conclusions

Considering the original problem presented in Section 1.1, we evaluated four existing visual regression testing tools in practice by conducting regression testing on a case application. In order to test the tools, we seeded the case application with visual appearance faults and executed regression tests with each tool. The experiment was conducted in a Continuous Integration environment. Next, we answer to the research questions.

Our evaluation findings present the advantages and disadvantages of each tool. All tools utilize the same "capture and compare" -method to test the visual regression. First, the tools capture the application UI state as a screenshot. After this, the screenshot is compared with a reference image taken during previous test runs. If a difference is found, the test execution fails.

The results indicate that two of the evaluated tools, PhantomCSS and Needle, are sufficient tools for automating the visual regression testing. The most significant advantage of PhantomCSS and Needle was their ability to interact with the application and to test individual elements instead of verifying the entire page at once. Thus the tools were able to test the entire application. The third tool, Depicted, provided a web UI to visualize the test results and to update the reference images easily. In comparison, Huxley did not provide any additional advantages to the other tools.

The evaluation revealed three common problem areas regarding to the visual regression testing. First, the reference images are difficult to update with Needle, PhantomCSS, and Huxley. In order to update the reference images, the developer has to remove the existing images and execute the tests once to capture new references. Second, the interoperability between different environments and browsers was found to be a problem when comparing screenshots. Finally, we encountered false positive test results with Huxley and Needle. The false positives distract the workflow and complicate

the testing process.

Using the evaluation findings as a basis, we combined the advantages of Needle and Depicted and implemented a new testing tool, Giffidiffi, to perform the visual regression testing. Giffidiffi consists of three components: server, user interface, and test runner. The server stores the test results together with the reference images. Web UI is used to verify the test results and to manage the reference images. To execute the tests and to capture the UI state using screenshots, we implemented a test runner using Python and Selenium WebDriver. To overcome the interoperability issues, the test runner is decoupled from the reference images, which allows different organizations to configure their test setup. The decoupling also allows the test runner to interact with the page before capturing the screenshot and sending it to the Giffidiffi server.

Giffidiffi has several improvements compared with the other tools. The most significant advantage is its ability to support the developers' workflow. The Giffidiffi UI provides clear visualizations of the test results and allows developers to manage reference images and to verify changes to the tested application.

Bibliography

[1] Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, W3C Recommendation 7 June 2011. http://www.w3.org/TR/CSS2/. Accessed 12.12.2013.

[2] CasperJS: a navigation scripting and testing utility for PhantomJS and SlimerJS, written in JavaScript. http://casperjs.org/. Accessed 30.12.2013.

[3] CSS Critic: A lightweight framework for regression testing of Cascading Style Sheets. https://github.com/BBC-News/wraith. Accessed 23.10.2013.

[4] Depicted: make continuous deployment safe by comparing before and after webpage screenshots for each release. https://github.com/bslatkin/dpxdt. Accessed 30.12.2013.

[5] Document Object Model. http://www.w3.org/DOM/. Accessed 20.11.2013.

[6] HTML 4.01 Specification, W3C Recommendation 24 December 1999. http://www.w3.org/TR/html4/. Accessed 13.10.2013.

[7] HTML5 specification, W3C candidate recommendation 17 december 2012. http://www.w3.org/TR/2012/CR-html5-20121217/. Accessed 13.10.2013.

[8] Huxley: a test-like system for catching visual regressions in Web applications. https://github.com/facebook/huxley. Accessed 30.12.2013.

[9] Hypertext transfer protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html. Accessed 20.11.2013.

[10] Indexed database API, W3C Candidate Recommendation 04 July 2013. http://www.w3.org/TR/IndexedDB/. Accessed 13.10.2013.

[11] Jasmine: behaviour-driven development framework for testing JavaScript code. `http://pivotal.github.io/jasmine/`. Accessed 13.12.2013.

[12] Mocha: the fun, simple, flexible JavaScript test framework. `http://visionmedia.github.io/mocha/`. Accessed 13.12.2013.

[13] Mozilla Developer Network: About JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript`. Accessed 20.11.2013.

[14] Needle: Automated tests for your CSS. `https://github.com/bfirsh/needle`. Accessed 30.12.2013.

[15] Nose: unit test framework for Python. `https://nose.readthedocs.org/en/latest/`. Accessed 30.12.2013.

[16] PhantomCSS: Visual/CSS regression testing with PhantomJS. `https://github.com/Huddle/PhantomCSS`. Accessed 30.12.2013.

[17] PhantomJS: Scriptable Headless WebKit. `https://github.com/ariya/phantomjs/`. Accessed 30.12.2013.

[18] Selenium - Browser automation framework. `http://code.google.com/p/selenium/`. Accessed 18.2.2014.

[19] Web storage, W3C Recommendation 30 July 2013. `http://www.w3.org/TR/webstorage/`. Accessed 13.10.2013.

[20] WebDriver: W3C Working Draft 12 March 2013. `http://www.w3.org/TR/webdriver/`. Accessed 18.2.2014.

[21] Andrews, A. A., Offutt, J., and Alexander, R. T. Testing web applications by modeling with fsms. *Software & Systems Modeling 4*, 3 (2005), 326–345.

[22] Artzi, S., Dolby, J., Jensen, S., Moller, A., and Tip, F. A framework for automated testing of javascript web applications. In *2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 571–580.

[23] Chang, T.-H., Yeh, T., and Miller, R. C. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 1535–1544.

[24] Di Lucca, G., Fasolino, A., Faralli, F., and De Carlini, U. Testing web applications. In *International Conference on Software Maintenance, 2002. Proceedings* (2002), pp. 310–319.

[25] Di Lucca, G. A., and Fasolino, A. R. Testing web-based applications: The state of the art and future trends. *Information and Software Technology 48*, 12 (2006), 1172–1186.

[26] Duvall, P. M., Matyas, S., and Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, June 2007.

[27] Fowler, M., and Foemmel, M. Continuous integration. *Thought-Works* (2006). `http://www.martinfowler.com/articles/continuousIntegration.html` Accessed 21.11.2013.

[28] Hieatt, E., and Mee, R. Going faster: testing the web application. *IEEE Software 19*, 2 (2002), 60–65.

[29] Holopainen, J. Regressiotestauksen tehostaminen. *Pro Gradu, Tietojenkasittelytieteen laitos, Kuopion Yliopisto* (2004).

[30] Jazayeri, M. Some trends in web application development. In *Future of Software Engineering, 2007. FOSE '07* (2007), pp. 199–213.

[31] Leff, A., and Rayfield, J. Web-application development using the Model/View/Controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International* (2001), pp. 118–127.

[32] Leung, H. K. N., and White, L. Insights into regression testing [software testing]. In *, Conference on Software Maintenance, 1989., Proceedings* (1989), pp. 60–69.

[33] Leung, H. K. N., and White, L. A study of integration testing and software regression at the integration level. In *, Conference on Software Maintenance, 1990, Proceedings* (1990), pp. 290–301.

[34] Marchetto, A., Tonella, P., and Ricca, F. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on* (2008), pp. 121–130.

[35] Martin, R. C. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[36] Memon, A., Nagarajan, A., and Xie, Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice 17*, 1 (2005), 27–64.

[37] Memon, A., and Xie, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering 31*, 10 (2005), 884–896.

[38] Memon, A. M. GUI testing: Pitfalls and process. *IEEE Computer 35*, 8 (2002), 87–88.

[39] Memon, A. M. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability 17*, 3 (2007), 137–157.

[40] Memon, A. M., and Soffa, M. L. Regression testing of GUIs. *SIGSOFT Softw. Eng. Notes 28*, 5 (2003), 118–127.

[41] Mesbah, A., and van Deursen, A. Migrating multi-page web applications to single-page AJAX interfaces. In *11th European Conference on Software Maintenance and Reengineering, 2007. CSMR '07* (2007), pp. 181–190.

[42] Mesbah, A., and van Deursen, A. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 210–220.

[43] Myers, G. J., Sandler, C., and Badgett, T. *The art of software testing*. John Wiley & Sons, 2011.

[44] Paulson, L. D. Building rich web applications with ajax. *Computer 38*, 10 (2005), 14–17.

[45] Pohja, M. Web application user interface technologies.

[46] Ricca, F., and Tonella, P. Analysis and testing of web applications. In *Proceedings of the 23rd international conference on Software engineering* (2001), IEEE Computer Society, pp. 25–34.

[47] Runeson, P. A survey of unit testing practices. *IEEE Software 23*, 4 (2006), 22–29.

[48] Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. Web browser as an application platform: the lively kernel experience. Tech. rep., Sun Microsystems, Inc., Mountain View, CA, USA, 2008.

[49] Tarhini, A., Ismail, Z., and Mansour, N. Regression testing web applications. In *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on* (2008), IEEE, pp. 902–906.

[50] Whittaker, J. A. What is software testing? and why is it so hard? *Software, IEEE 17*, 1 (2000), 70–79.

[51] Xiaochun, Z., Bo, Z., Juefeng, L., and Qiu, G. A test automation solution on GUI functional test. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on* (2008), pp. 1413–1418.